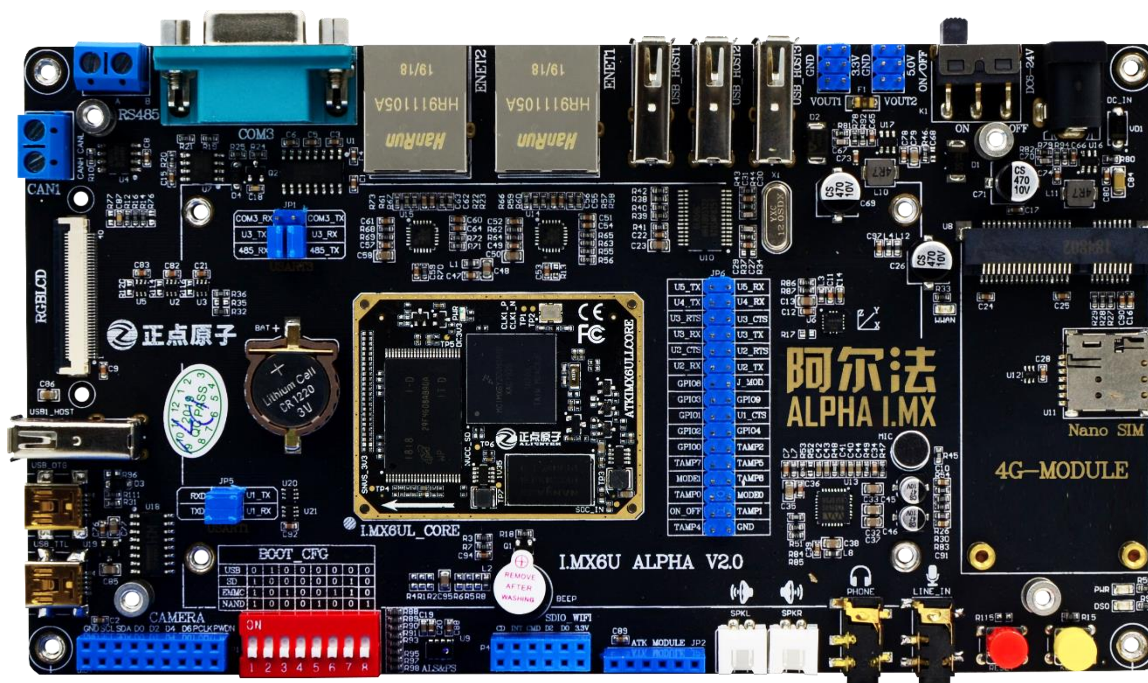
I.MX6U 嵌入式 Linux 驱动开发指南 V1.0

-正点原子 I.MX6U ALPHA 开发板教程



正点原子广州市星翼电子科技有限公司

淘宝店铺 1：http://eboard.taobao.com

淘宝店铺 2：http://openedv.taobao.com

技术支持论坛（开源电子网）：www.openedv.com

原子哥在线教学：www.yuanzige.com

官方网站：www.alientek.com

最新资料下载链接：http://www.openedv.com/posts/list/13912.htm

E-mail: 389063473@qq.com　　**QQ:** 389063473

咨询电话：020-38271790

传真号码：020-36773971

团队：正点原子团队

正点原子，做最全面、最优秀的嵌入式开发平台软硬件供应商。

友 情 提 示

如果您想及时免费获取"正点原子"最新资料，敬请关注正点原子微信公众平台，我们将及时给您发布最新消息和重要资料。



关注方法：

(1)微信"扫一扫"，扫描右侧二维码，添加关注

(2)微信→添加朋友→公众号→输入"正点原子"→关注

(3)微信→添加朋友→输入"alientek_stm32"→关注



文档更新说明

版本	版本更新说明	负责人	校审	发布日期
V1.0	初稿：	左忠凯	左忠凯	2019.10.26

目录

前言

当今社会是一个电子信息技术飞速发展的年代，小到玩具，家电，大到手机、飞机、潜艇等，都离不开电子信息技术。社会上对于电子信息方面的人才需求也很大，从就业的角度而言电子、计算机、通信以及信息方面专业的毕业生工资都比较高，就业也比较容易，尤其是嵌入式、Linux 和 Android 等相关开发工作。

本书主要讲解嵌入式 Linux 中的驱动开发，也会涉及到裸机开发的内容，相信大部分读者和作者一样，以前都是做单片机开发的工作，比如 51 或者 STM32 等。单片机开发很难接触到更高层次的系统方面的知识，单片机用到的系统都很简单，比如 UCOS、FreeRTOS 等等，这些操作系统都是一个 kernel，如果需要网络、文件系统、GUI 等这些就需要开发者自行移植。而移植又是非常痛苦的一件事情，而且移植完成以后的稳定性也无法保证。即使移植成功以后后续的开发工作也比较繁琐，因为不同的组件其 API 操作函数都不同，没有一个统计的标准，使用起来的话学习成本比较高。这个时候一个功能完善的操作系统显得尤为重要：具有统一的标准；提供完善的多任务管理、存储管理、设备管理、文件管理和网络等。Linux 就是这样一个系统，当然这样的系统还有很多，比如 Windows，MacOS，UNIX 等等。本书我们讲解 Linux，而 Linux 开发可以分为底层驱动开发和应用开发，本书讲解的是 Linux 的驱动开发，主要面向与那些此前使用 STM32 的开发者。平心而论，如果此前只会 51 单片机开发的话我是非常不建议直接上手 Linux 驱动开发的，因为 51 和 Linux 驱动开发的差距太大了！笔者建议在学习嵌入式 Linux 驱动开发之前一定要学一下 STM32 这种 Cortex-M 内核的 MCU，因为 STM32 这样的 MCU 其内部资源基本和可以运行 Linux 的 CPU 差不多，如果会 STM32 的话上手 Linux 驱动开发就会容易很多。笔者就是此前做了 4 年 STM32 开发工作，然后转的 Linux 驱动开发，整个过程比较顺畅。

鉴于当前 STM32 非常火爆，学习者众多，如何帮助 STM32 学习者顺利的转入 Linux 驱动开发是笔者思考了很久的问题。为此笔者本书和相应例程的安排如下：

1、选取合适的 CPU

理论上来讲，如果 ST 公司有可以运行的 Linux 的芯片那再好不过了，因为大家对 STM32 很熟悉，但是在写本书的时候 ST 也没有可以运行 Linux 的 CPU。Linux 驱动开发入门的 CPU 一定不能复杂！！！比如像三星的 Exynos 4412、Exynos 4418 等这些手机 CPU，这些 CPU 性能很强大，带有 GPU、支持硬件视频解码、可以运行 Andriod。但是正是它们的性能过于强大，功能过于繁杂，所以不适合 Linux 驱动开发入门。一款外设和 STM32H7 这样的 MCU 差不多的 CPU 就非常适合 Linux 入门，三星的 2440 就非常合适，但是 2440 早已停产了，学了以后工作上肯定又用不到了，又得学习其他的 CPU，有点浪费时间。作者花了不少时间终于找到了一款合适的 CPU，那就是 NXP 的 I.MX6UL！可以认为 I.MX6UL 就是一款可以跑 Linux 的 STM32，外设功能和 STM32 相似，如果此前学习过 STM32 的话会非常容易上手 I.MX6UL。而且目前 I.MX6UL 正在大量出货，这是一款工业级的 CPU，大量的以前三星 2440、6410 做的产品更新换代的绝佳之选，学习完 I.MX6UL 以后工作就可以直接使用了。本书选取开发平台为正点原子的 I.MX6U-ALPHA 开发板，其他厂商的 I.MX6UL 开发板也可以参考本书。

2、开发环境讲解

STM32 的开发都是在 Windows 系统下进行的，使用 MDK 或者 IAR 这样的集成 IDE，但是嵌入式 Linux 驱动开发需要的主机是 Linux 平台的，也就是你必须先在自己的电脑上安装 Linux 系统，Linux 系统发行版有 Ubuntu、CentOS、Fdeora、Debian 等等，本书我们使用 Ubuntu 操作系统。本书假设大家此前从来没有接触过 Ubuntu 操作系统，因此会有详细的 Ubuntu 操作

系统安装、使用教程，帮助大家数据开发环境。

3、合理的裸机例程

学习嵌入式 Linux 驱动开发建议大家先学习裸机开发(当然了，如果学习过 STM32 的话可以跳过裸机学习)，Linux 驱动开发非常庞大、繁琐。要想进行 Linux 驱动开发，必须要先移植 Uboot、然后移植 Linux 系统和根文件系统到你的开发平台上。而 Uboot 又是一个超大的裸机综合例程，因此如果你没有学习过裸机例程那么 Uboot 移植将会有点困难，尤其是当要修改 Uboot 代码的时候。做 STM32 开发的话基本都是裸机开发，在 IDE 平台下编写代码，可以使用 ST 提供的库。但是在 Ubuntu 下编写 I.MX6UL 裸机例程的话就没有这么方便了，没有 MDK 和 IAR 这样的 IDE，没有 ST 提供的库。所有的一切都需要我们自己搭建，大家不用担心，本书包括视频会有详细的讲解。在裸机例程内容方面，我们提供了数十个裸机例程，由浅入深，涵盖了大部分常用的功能，比如 IO 输入和输出、中断、串口、定时器、DDR、LCD、I2C 等。学习完裸机例程以后基本就对 I.MX6UL 这颗 CPU 非常了解了。再去学习 Linux 驱动开发的话就很轻松了。

4、Uboot、Linux 和根文件系统移植

学习完裸机例程以后就是 Linux 驱动开发了，但是在进行 Linux 驱动开发之前要先在使用的开发板平台上移植好 Uboot，Linux 和根文件系统。这是 Linux 驱动开发的第一个拦路虎，因此本书和相应的视频会着重讲解 Uboot/Linux 和根文件系统的移植。

5、嵌入式 Linux 驱动开发

当我们把 Uboot，Linux 和根文件系统都在开发板上移植好了以后就可以开始 Linux 驱动开发了。Linux 驱动有三大类:字符设备驱动、块设备驱动和网络设备驱动，这三大类我们都会详细的讲解，并且配有数十个相应的例程，由简入深，从最简单的点灯，到最后的网络设备驱动。

本书一共分四篇，每篇对应一个不同的阶段：

第一篇：Ubuntu 操作系统入门

本篇主要讲解 Ubuntu 操作系统的使用，本篇不涉及到任何嵌入式方面的知识，全部是在 PC 上完成的，只要安装好 Ubuntu 操作系统即可。

第二篇： ARM 裸机开发

从本篇开始我们就正式开始使用开发板学习了，本篇通过数十个裸机例程来帮助大家了解 I.MX6UL 这颗 CPU。为以后的 Linux 驱动开发做准备，通过本篇大家可以掌握在 Ubuntu 下进行 ARM 开发的方法。

第三篇：Uboot、Linux 和根文件系统移植

本篇讲解如何将 Uboot、Linux 和根文件系统移植到我们的开发板上，为后面的 Linux 驱动开发做准备。

第四篇： Linux 驱动开发

前面做了那么多的工作就是为了本篇，因此本篇注定了将是本书重中之重，大家应该投入精力最多的一篇，望大家做好准备。

通过上面四篇的学习，大家基本掌握了嵌入式 Linux 驱动的开发流程，本书旨在引导大家入门 Linux 驱动开发，更加深入的研究就需要大家自行查阅其他更加专业的书籍了，祝愿大家学习顺利。

第一篇 Ubuntu 系统入门篇

本篇主要讲解 Ubuntu 系统，包括如何在虚拟机上安装 Ubuntu 操作系统，安装好以后 Ubuntu 的设置、基本操作等等。在正式进行嵌入式开发之前肯定是要先学会 Ubuntu 系统如何使用的，由于 Ubuntu 系统功能很庞大，如果要详细讲解的话一本书都讲不完，因此本篇只做 Ubuntu 系统入门讲解，掌握我们进行嵌入式开发所需的技能即可，如果想详细的学习 Ubuntu 系统的话可以参考其他的书籍，比如经典的《鸟哥的 linux 私房菜》，《鸟哥的 linux 私房菜》这本书使用的 CentOS 操作系统，但是 Ubuntu 下完全可以使用。

当 Ubuntu 系统入门以后，我们重点要学的就是如何在 Linux 下进行 C 语言开发，如何使用 gcc 编译器、如何编写 Makefile 文件等等。

如果此前已经使用过 Ubuntu 操作系统，并且从事过 Linux C 编程工作的话本篇就不需要看了，可以直接跳到第二篇，开始 ARM 裸机开发篇的学习。

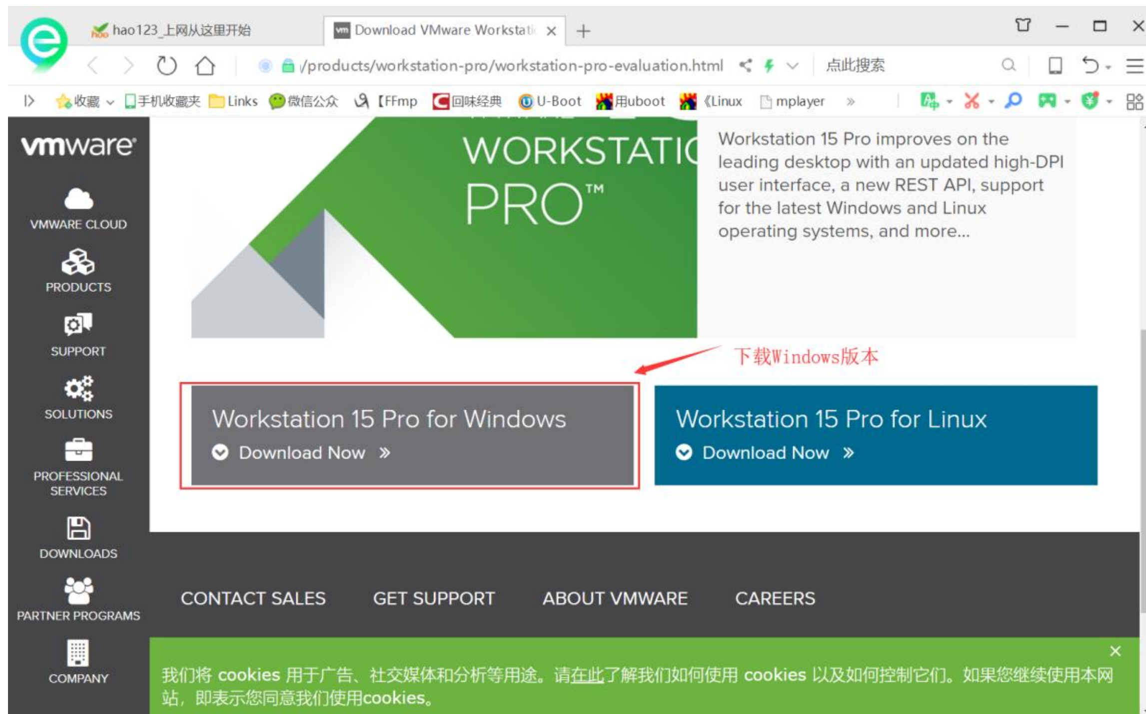
第一章 Ubuntu 系统安装

Linux 的开发需要在 Linux 系统下进行，这就要求我们的 PC 主机安装 Linux 系统，本书我们选择 Ubuntu 这个 Linux 发行版系统。本章讲解如何安装虚拟机，以及如何在虚拟机中安装 Ubuntu 系统，安装完成以后如何做简单的设置。如果已经对于虚拟机以及 Ubuntu 基础操作已经熟悉的话就可以跳过本章。

1.1 安装虚拟机软件 VMware

不是安装 Ubuntu 吗？怎么要先安装虚拟机呢？虚拟机是个啥？相信大部分第一次安装 Ubuntu 的人都会有这个疑问。我不能直接安装 Ubuntu 吗？能不能不要虚拟机呢？答案是肯定可以的！直接在电脑上安装 Ubuntu 以后你的电脑就是一个真真正正的 Ubuntu 电脑了，你可以再安装一个 Windows 系统，这样你的电脑就是双系统了，在开机的时候可以选择不同的系统启动。但是这样的话会有一个问题，那就是你每次只能选择其中的一个系统启动，要么 Windows 要么 Ubuntu，但是我们再开发的时候很多时候需要再 Windows 和 Ubuntu 下来回切换，Windows 系统下的软件资源要比 Ubuntu 下丰富的多，比如我们在 Windows 用 Source Insight 这个神器编写代码，然后拿到 Ubuntu 下编译。这个就涉及到两个系统切换问题，显然如果你直接在电脑上安装 Ubuntu 以后就没法做到，因为你每次开机只能在 Windows 和 Ubuntu 下二选一。
如果 Ubuntu 系统能作为 Windows 下的一个软件就好了，我们默认启动 Windows 系统，需要用到 Ubuntu 的话直接打开这个软件就行了。当然可以！这个就要借助虚拟机了，虚拟机顾名思义就是虚拟出一个机器，然后你就可以在这个机器上安装任何你想要的系统，相当于在克隆出一个你的电脑，这样在主机上运行 Windows 系统，当我们需要用到 Ubuntu 的话就打开安装有 Ubuntu 系统的虚拟机。

虚拟机的实现我们可以借助其他软件，比如 Vmware Workstation，Vmware Workstation 是收费软件，免费的虚拟机软件有 Virtualbox。本书我们使用 Vmware Workstation 软件来做虚拟机。Vmware Workstation 软件可以在 Wmeare 官网下载，下载地址：https://www.vmware.com/products/workstation-pro/workstation-pro-evaluation.html，当前最新的版本是 Vmware Workstation Pro 15，我们下载 Windows 版本的，如下图所示：



我们已经在开发板光盘里面提供了 Vmware Workstation 软件，大家可以直接使用，在光盘目录：开发板光盘->3、软件->VMware-workstation-full-15.5.0-14665864.exe。WMware Workstation 的安装和普通软件安装一样，双击 VMware-workstation-full-15.5.0-14665864.exe 进入安装界面，如图 1.1.1 所示：



图 1.1.1 VMware 安装界面

点击图 1.1.1 中的“下一步”，进入图 1.1.2 所示步骤：

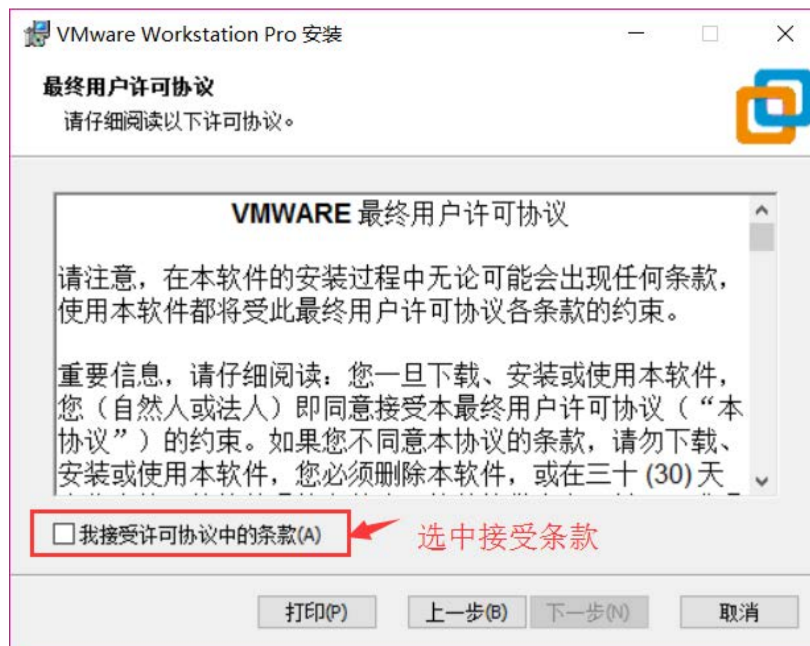


图 1.1.2 VMware 条款

先选择图 1.1.2 中的“我接受许可协议中的条款”，然后在选择“下一步”，进入图 1.1.3 所示步骤：

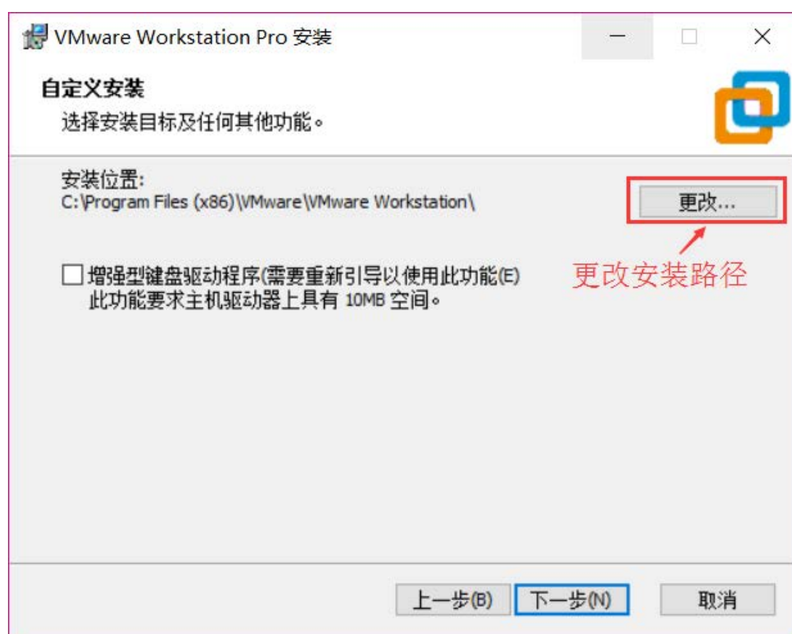


图 1.1.3 选择安装路径

图 1.1.3 中选择软件的安装路径，点击“更改”按钮，然后根据自己的实际需要选择合适路径即可,我的安装路径如图 1.1.4 所示：

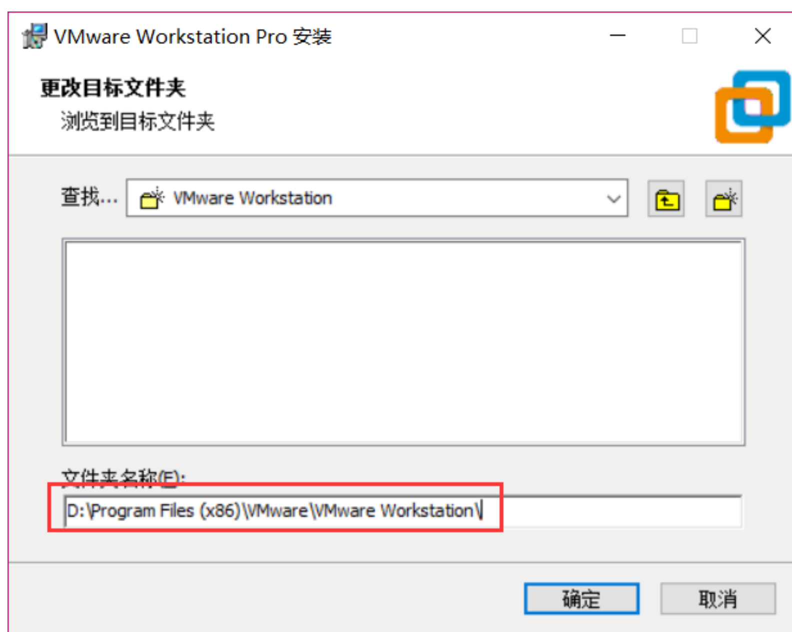


图 1.1.4 安装路径

选择好路径以后点击图 1.1.4 中的“确定”按钮，然后回到图 1.1.3 所示界面，点击图 1.1.3 中的“下一步”，进入图 1.1.5 所示界面：

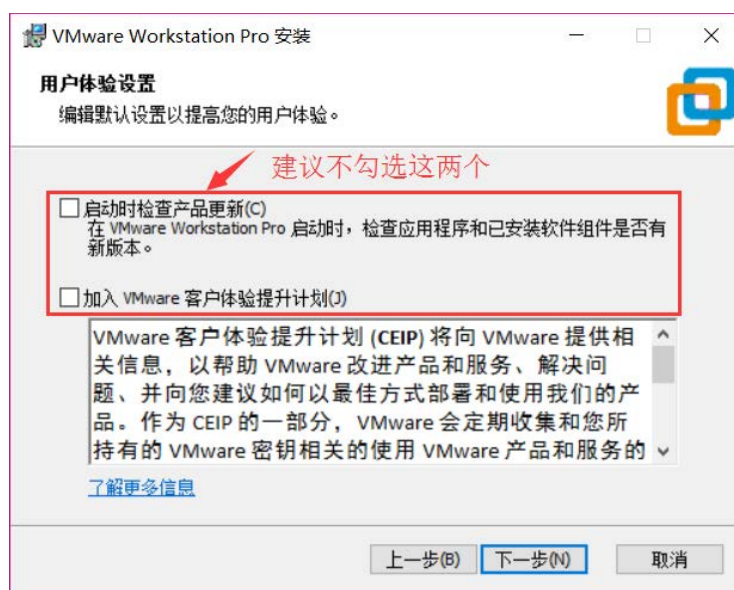


图 1.1.5 检查更新界面

在图 1.1.5 中，会有两个复选框，默认都是选中的，建议不要选中！然后点击图 1.1.5 中的“下一步”按钮，进入图 1.1.6 所示界面：

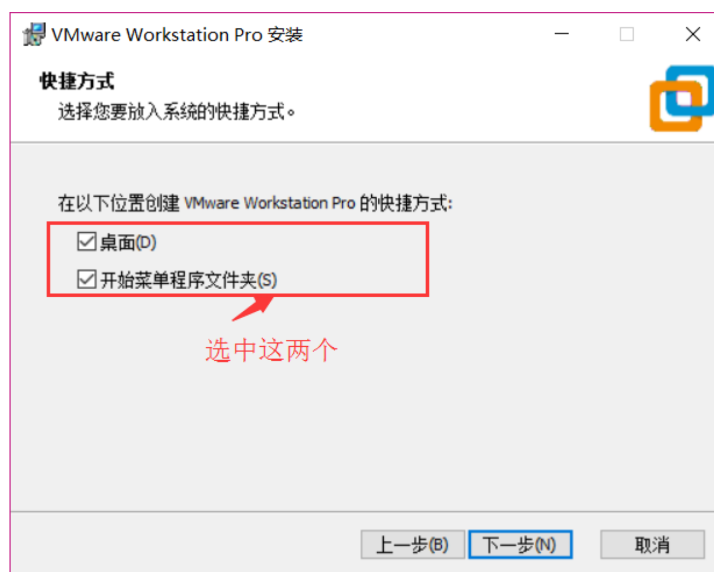


图 1.1.6 快捷方式设置

在图 1.1.6 中有两个选项，我们都选中，这样在安装完成以后就会在开始菜单和桌面上有 VMware 的图标，选中以后点击图 1.1.6 中的“下一步”，进入图 1.1.7 界面：

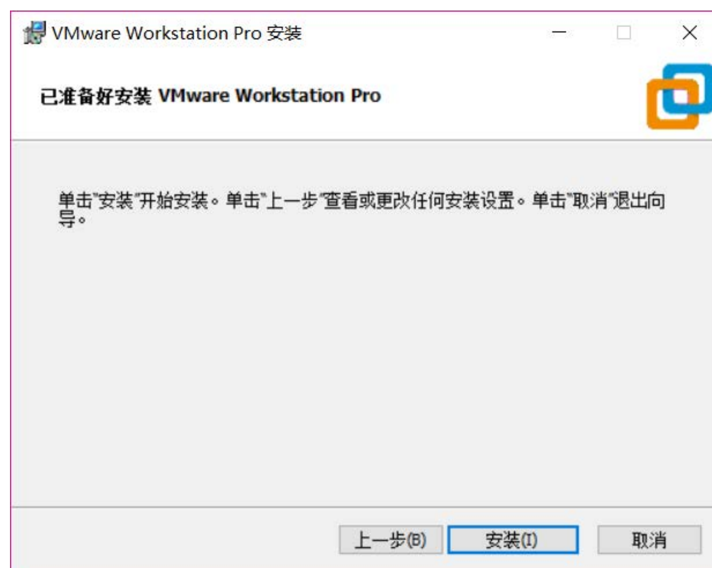


图 1.1.7 安装确定界面

前面几步已经设置好安装参数了，如果许需要修改安装参数的话就点击图 1.1.7 中的“安装”按钮开始安装 VMware，安装过程如图 1.1.8 所示：

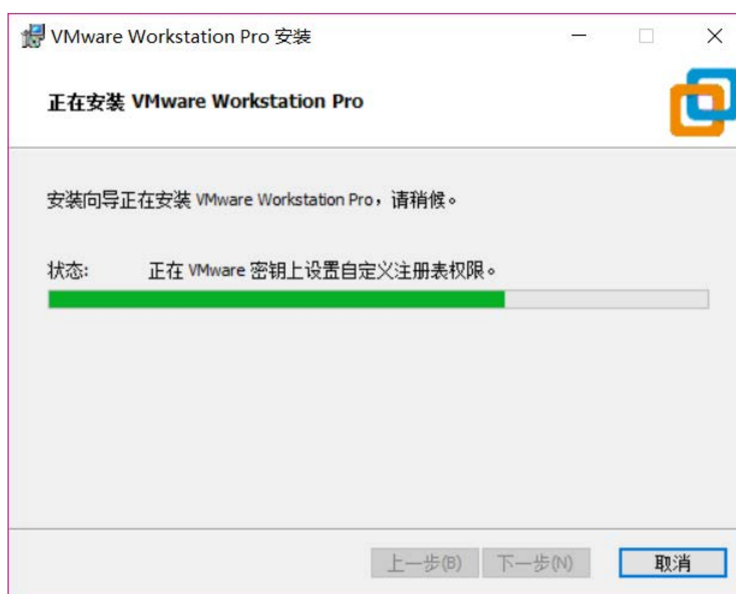


图 1.1.8 安装进行中

图 1.1.8 就是安装过程，耐心等待几分钟，等待安装完成，安装完成以后会有如图 1.1.9 所示提示：



图 1.1.9 安装完成

点击图 1.1.9 中的“完成”按钮，完成 VMware 的安装，安装完成以后就会在桌面上出现 VMware Workstation Pro 的图标，如图 1.1.10 所示：

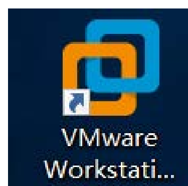


图 1.1.10 VMware 桌面图标

双击图 1.1.10 中的图标打开 VMware 软件，在第一次打开软件的时候会提示你输入许可证密钥，如图 1.1.11 所示：

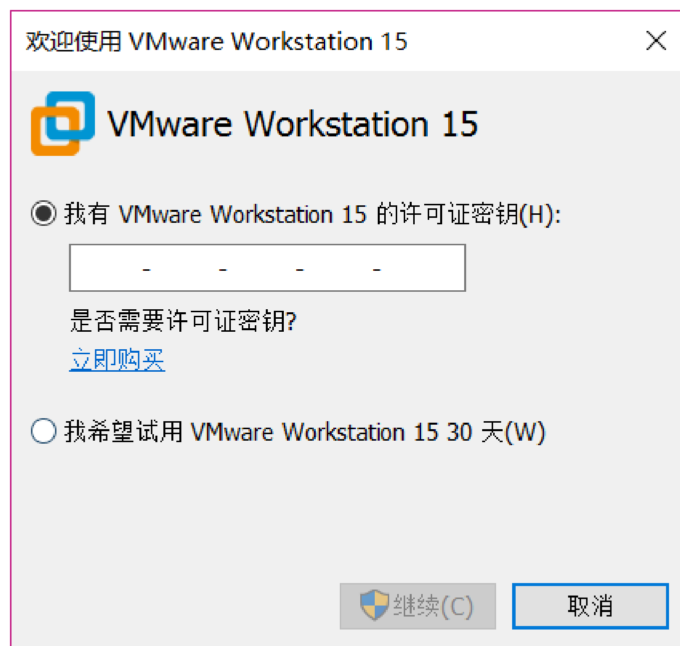


图 1.1.11 输入许可证密钥

前面说了 VMware 是付费软件，是需要购买的，如果你购买了 VMware 的话就会有一串许可密钥，如果没有购买的话就选择“我希望试用 VMware Workstation 15 30 天”选项，这样你就可以体验 30 天 VMware。输入密钥以后点击“继续按钮”，如果你的密钥正确的话就会提示你购买成功，如图 1.1.12 所示：

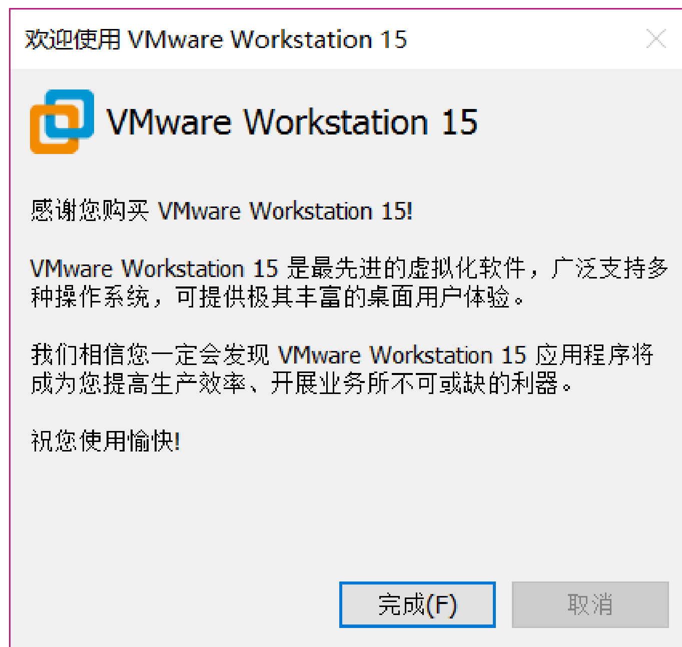


图 1.1.12 购买 VMware 成功

点击图 1.1.12 中的“完成”按钮，VMware 软件正式打开，界面如图 1.1.13 所示：

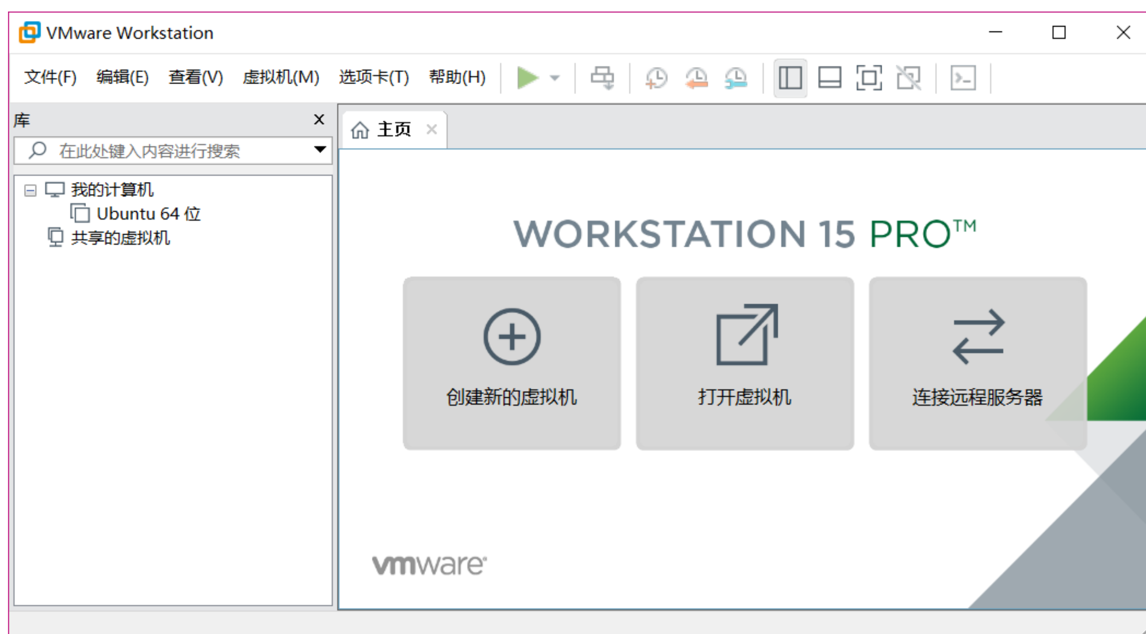


图 1.1.13 VMware Workstation 主界面

至此，虚拟机软件 VMware 安装成功。

1.2 创建虚拟机

安装好 VMware 以后我们就可以在 VMware 上创建一个虚拟机，打开 VMware，选择：文件->新建虚拟机，如图 1.2.1 所示：

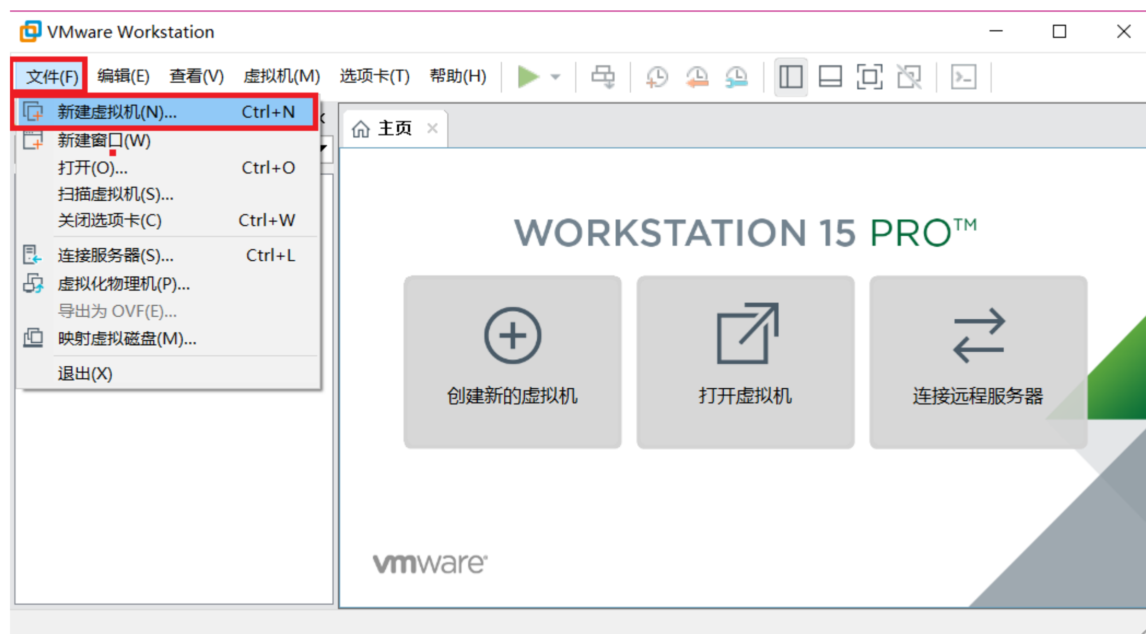


图 1.2.1 新建虚拟机

打开图 1.2.2 所示创建虚拟机向导界面：

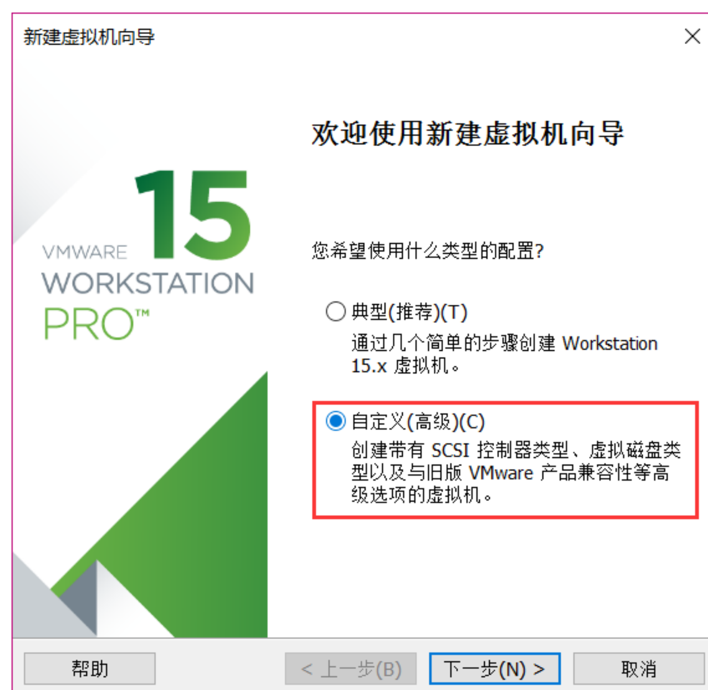


图 1.2.2 创建虚拟机向导

选中图 1.2.2 中的“自定义”选项，然后选择“下一步”，进入图 1.2.3 所示硬件兼容性选择界面：

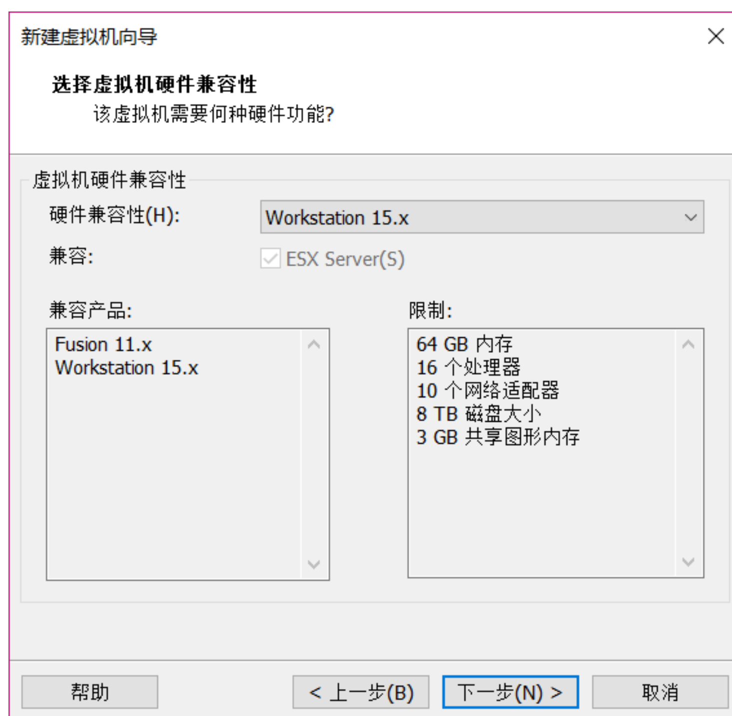


图 1.2.3 硬件兼容性选择

在图 1.2.3 中我们使用默认值就行了，直接点击“下一步”，进入图 1.2.4 所示的操作系统安装界面：

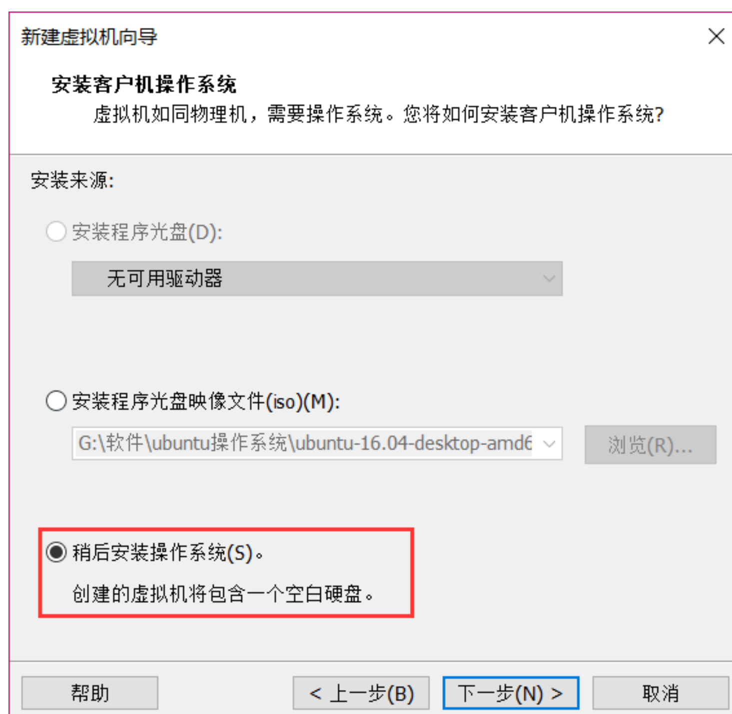


图 1.2.4 安装客户机操作系统

图 1.2.4 就是选择你新创建的虚拟机要安装什么系统？windos 还是 linux，如果你要现在就安装系统的话需要准备好系统文件，一般是.iso 文件。我们现在不安装系统，因此选择“稍后安装操作系统(S)”这个选项，然后选择“下一步”，进入图 1.2.5 所示界面：

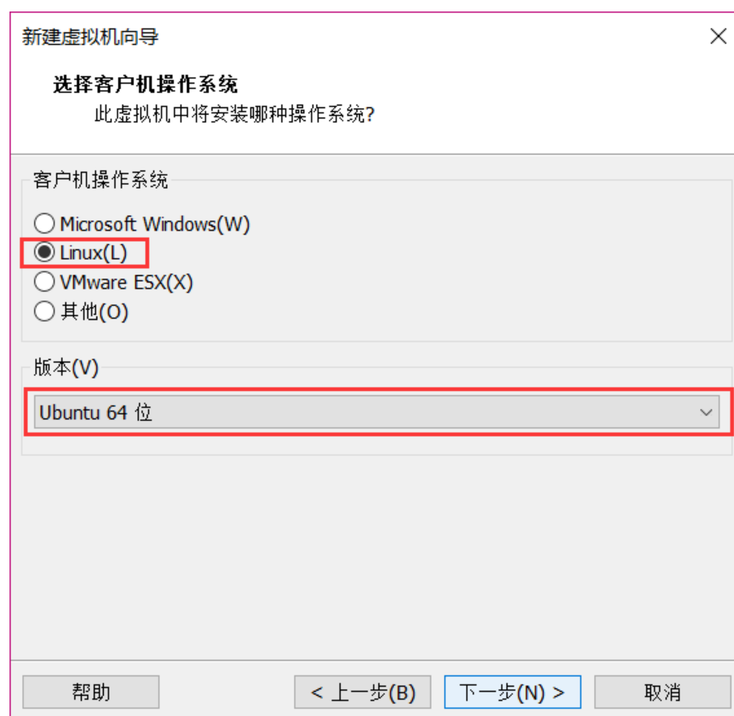


图 1.2.5 客户机操作系统选择

图 1.2.5 中依旧是让你选择你要在虚拟机中装什么系统，图 1.2.5 是和图 1.2.4 配合在一起使用的，在图 1.2.4 中放入系统文件(.iso 文件)，然后在图 1.2.5 中选择你图 1.2.4 中放入的是什么系统，然后 VMware 就会稍后自动安装所设置的系统。在图 1.2.4 中我们没有设置系统文件，因此图 1.2.5 是没用的，不过我们还是在图 1.2.5 中的客户机操作系统一栏选择“Linux”，版本选择 Ubuntu 64 位，然后点击“下一步”，进入图 1.2.6 所示界面：

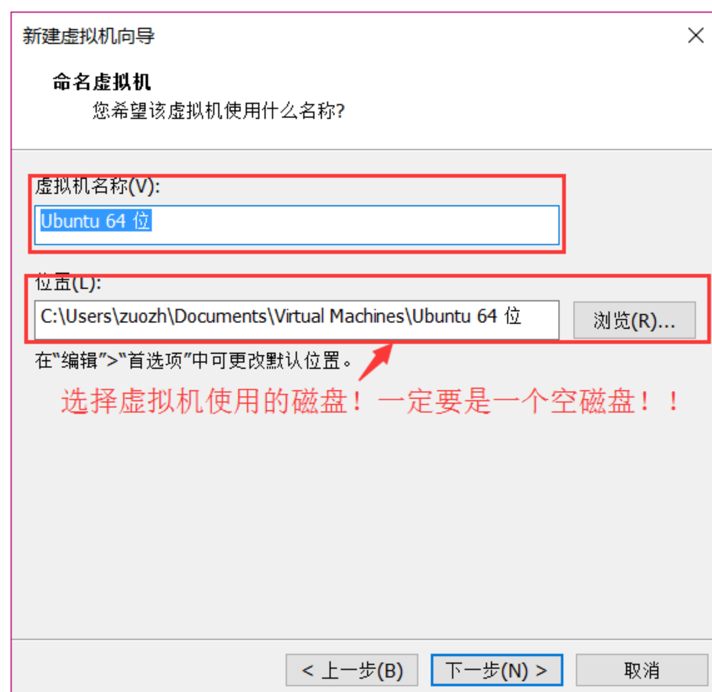


图 1.2.6 命名虚拟机

在图 1.2.6 中上面是命名虚拟机名字，大家可以根据自己的使用习惯给虚拟机命名，重点是

下面的虚拟机位置选择！我们要给虚拟机单独清理出一块磁盘，做嵌入式开发建议这块空磁盘的大小不小于 100GB，比如我清理除了一个 196GB 的 I 盘给虚拟机使用，如图 1.2.7 所示：

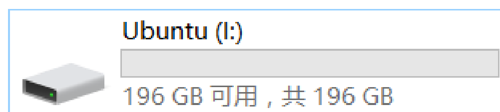


图 1.2.7 虚拟机所使用的磁盘

清理出虚拟机专用的磁盘以后然后就在图 1.2.6 中的位置出选择这个磁盘，比如我的位置选择如图 1.2.8 所示：

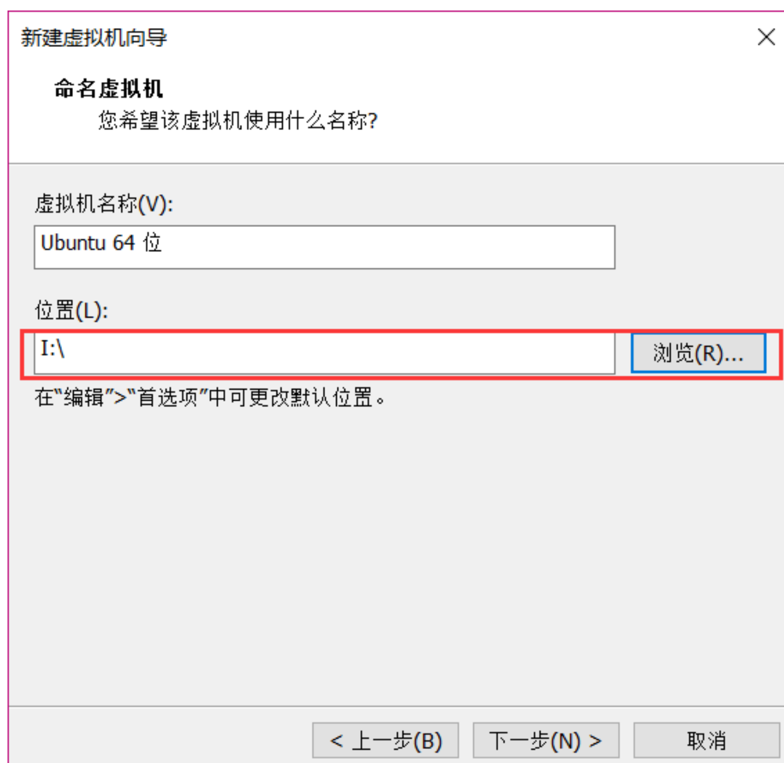


图 1.2.8 选择虚拟机磁盘位置

设置好图 1.2.8 中的虚拟机磁盘位置以后点击“下一步”，进入图 1.2.9 所示的处理器配置选择界面：

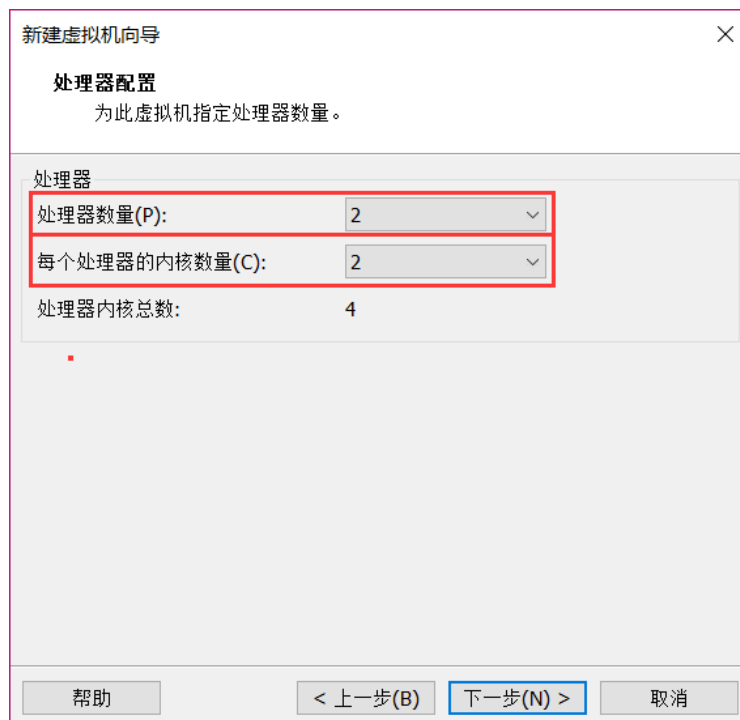


图 1.2.9 处理器配置界面

图 1.2.9 中就是配置你的虚拟机所使用的处理器数量，以及每个处理器的内核数量，这个要根据自己实际使用的电脑 CPU 配置来设置。比如我的电脑 CPU 是 I7-4720HQ，这是个 4 核 8 线程的 CPU，因此我就可以分 2 个核给 VMware，然后 I7-4720HQ 每个物理核有两个逻辑核，因此每个处理器的内核数量就是 2，所以的 VMware 虚拟机配置就如图 1.2.9 所示，大家根据自己的实际电脑 CPU 配置来设置即可，设置好以后点击“下一步”，进入图 1.2.10 所示内存配置界面：

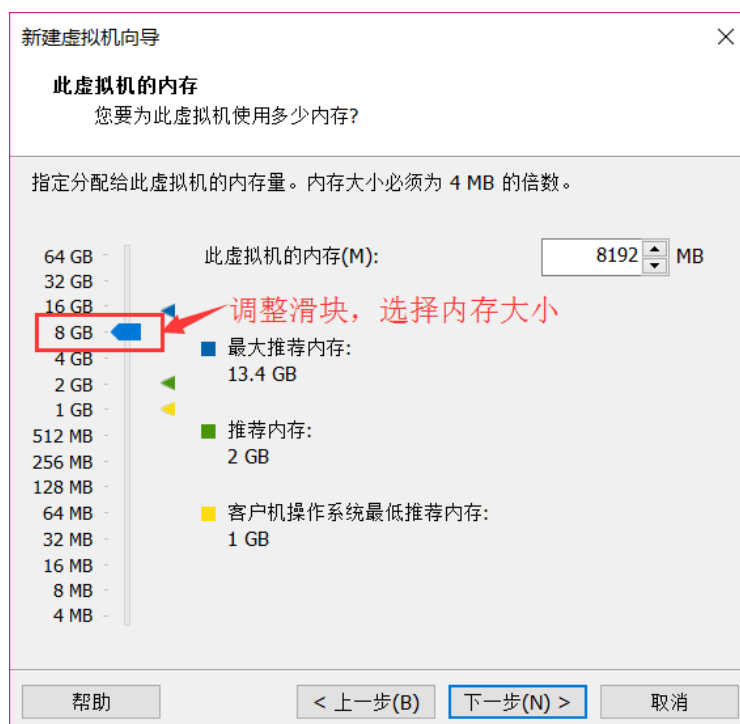


图 1.2.10 内存配置

同样的在图 1.2.10 中根据自己电脑的实际内存配置来设置分给虚拟机的内存大小，比如我的电脑是 16GB 的内存，因此我可以给虚拟机分配 8GB 的内存。配置好虚拟机的内存大小以后点击“下一步”，进入图 1.2.11 所示的网络类型选择界面：

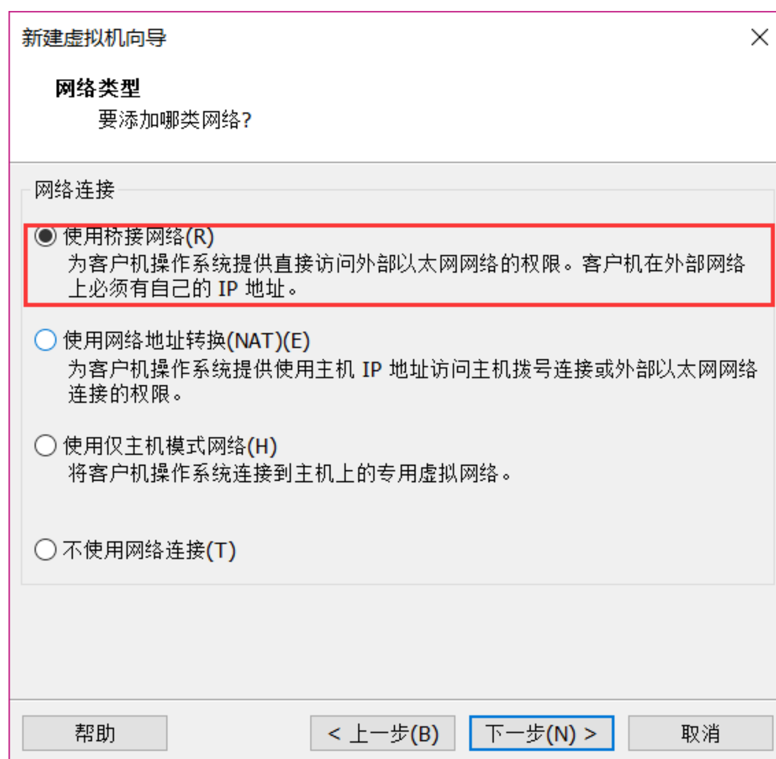


图 1.2.11 网络类型选择界面

在图 1.2.11 中我们选择“使用桥接网络”，然后点击“下一步”，进入图 1.2.12 所示的选择 I/O 控制器类型界面：

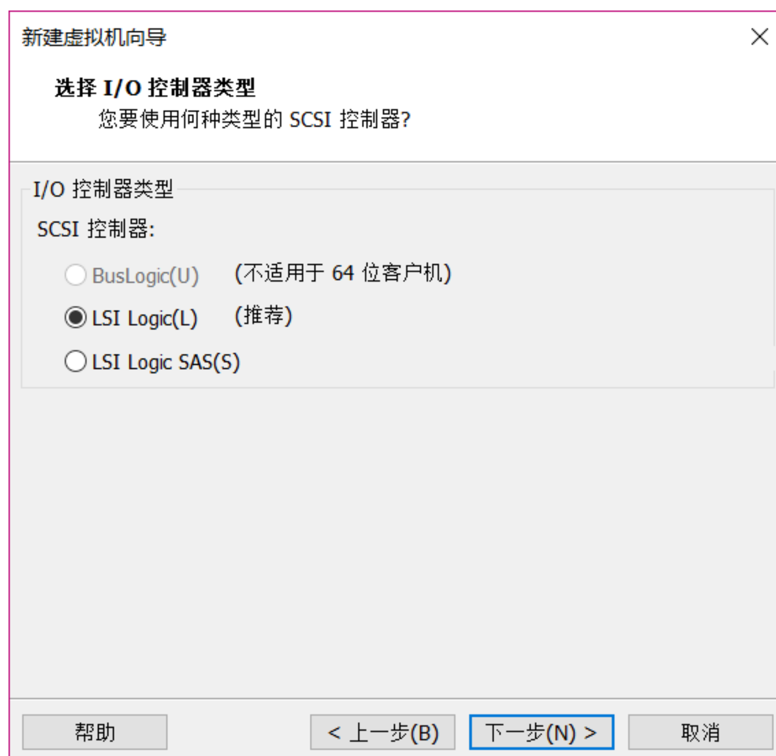


图 1.2.12 I/O 控制器选择

I/O 控制器类型选择默认值就行，也就是“LSI Logic”，然后点击“下一步”，进入磁盘类型选择界面，如图 1.2.13 所示：



图 1.2.13

图 1.2.13 中选择磁盘类型，使用默认值“SCSI”即可，然后点击“下一步”，进入选择磁盘

界面，如图 1.2.14 所示：

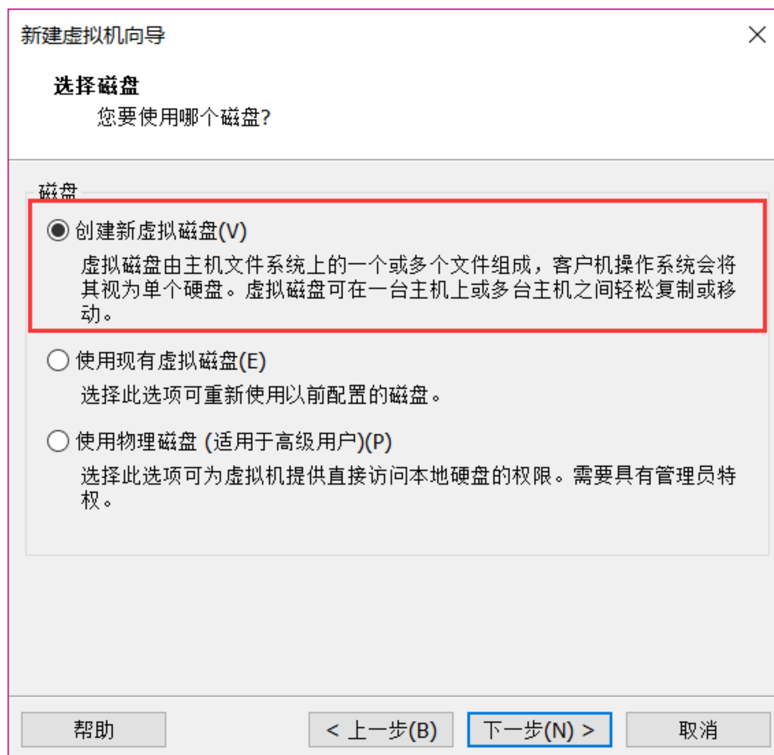


图 1.2.14 磁盘选择

图 1.2.14 中使用默认值，即“创建新虚拟磁盘”，这样我们前面设置好的那个空的磁盘就会被创建为一个新的磁盘，设置要以后点击“下一步”，进入磁盘容量设置界面，如图 1.2.15 所示：

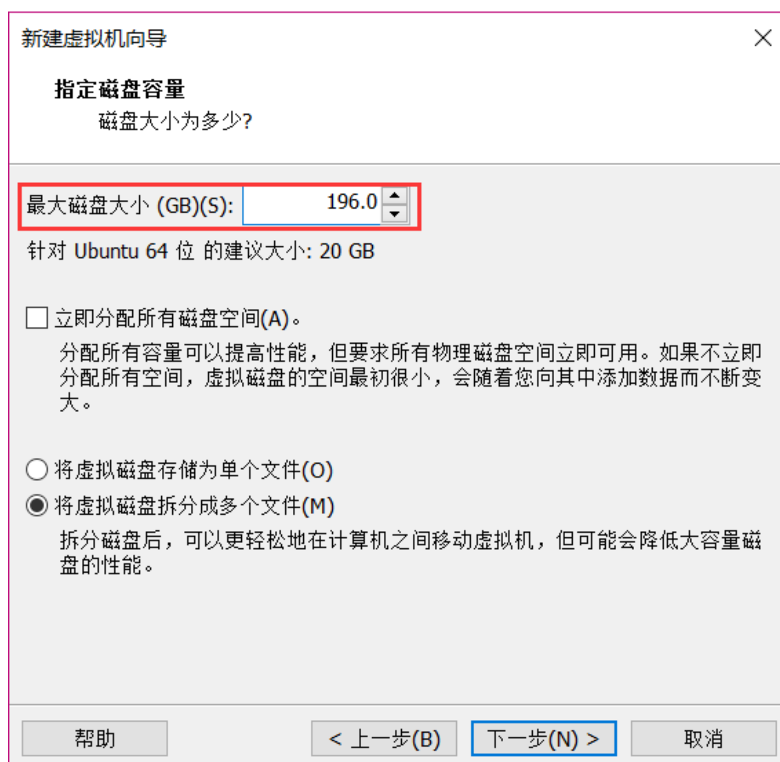


图 1.2.15 磁盘容量设置

图 1.2.15 是用来设置我们清出的空的磁盘多少是给虚拟机用的，我们清出了一个空磁盘肯定是全部给虚拟机用的，因此设置最大磁盘大小为空磁盘的大小，比如图 1.2.7 中我的那个 I 盘是 196GB 的，因此图 1.2.15 中就设置最大磁盘大小为 196GB，然后点击“下一步”，进入图 1.2.16 所示界面指定磁盘文件，

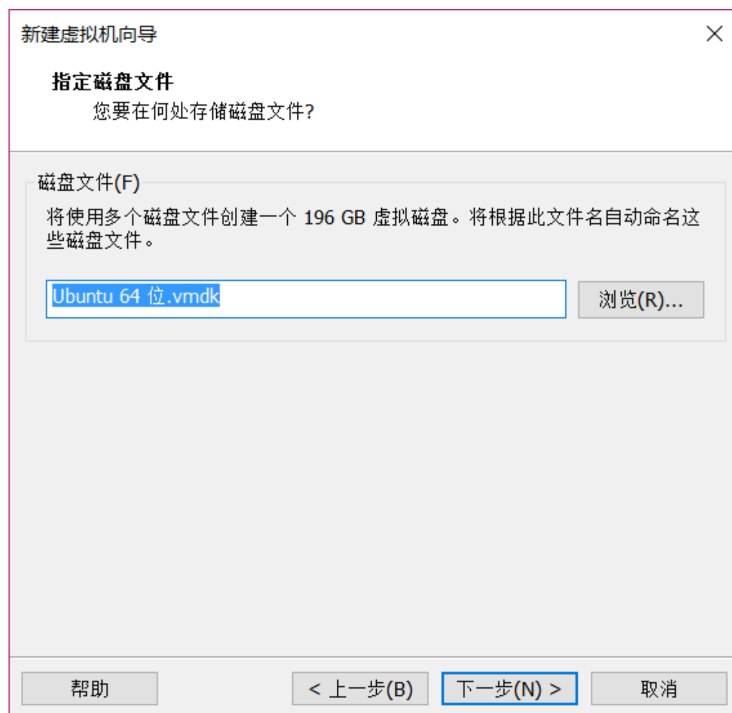


图 1.2.16 指定磁盘文件

图 1.2.16 使用默认设置，不要做任何修改，直接点击“下一步”，进入已准备好创建虚拟机界面，如图 1.2.17 所示：

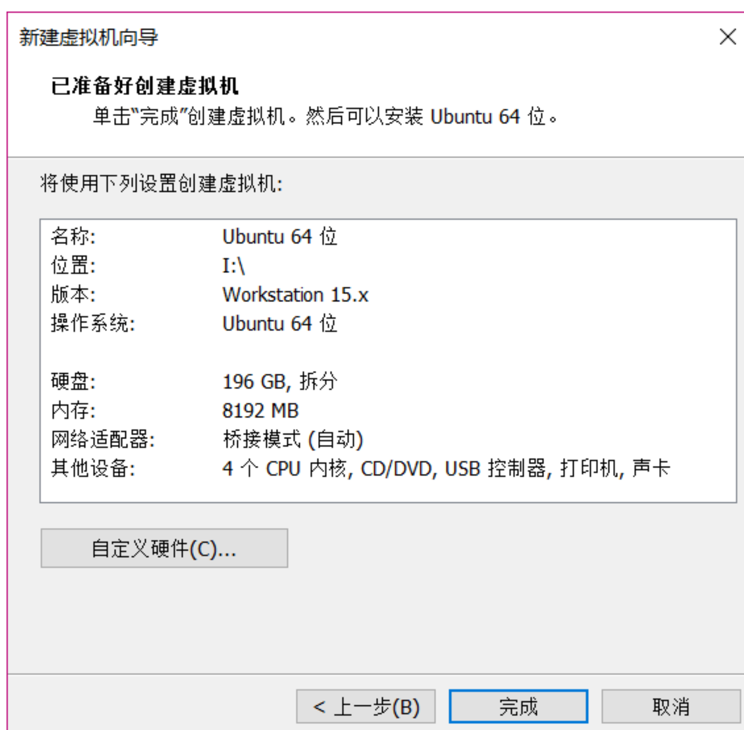


图 1.2.17 准备创建虚拟机

在图 1.2.17 中确认自己的虚拟机配置，如果确认无误就点击“完成”，如果有误的话就返回有误的配置界面做修改，点击“完成”按钮以后就会创建一个虚拟机，如图 1.2.17 所示：

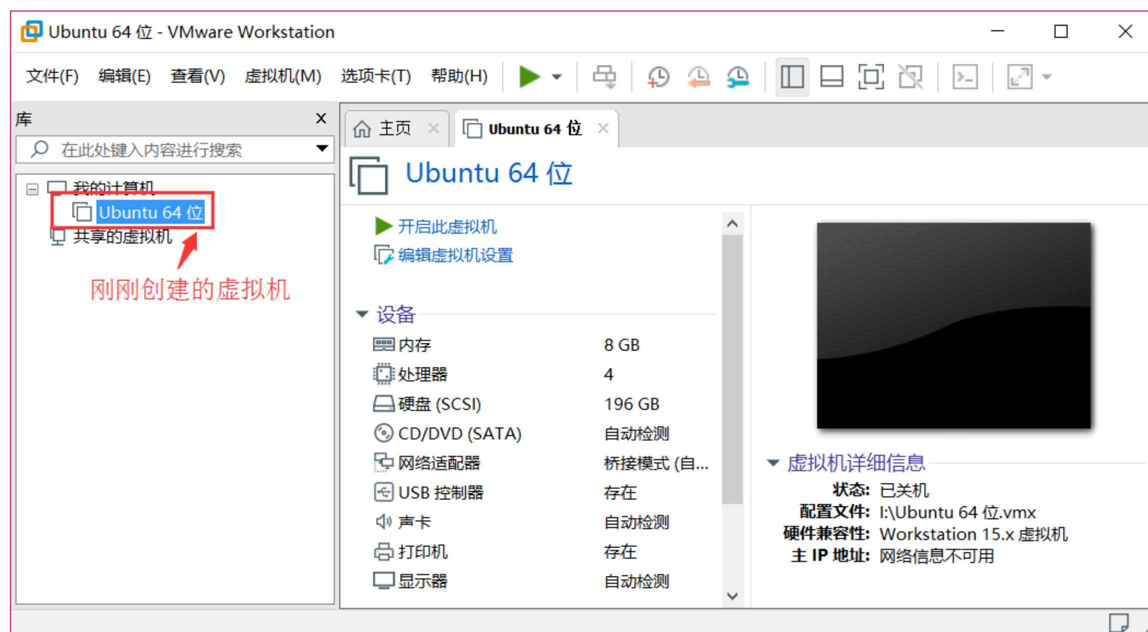


图 1.2.17 新创建的虚拟机

创建虚拟机成功以后就会在右侧的：我的计算机下出现刚刚创建的虚拟机“Ubuntu 64 位”，点击一下就会在右侧打开这个虚拟机的详细信息，如图 1.2.18 所示：



图 1.2.18 新建虚拟机配置信息

在图 1.2.18 中的设备一栏我们可以看到虚拟机详细的配置信息，图 1.2.19 所示的两个按钮就是虚拟机的开关，



图 1.2.19 虚拟机开关

图 1.2.19 中的这两个绿色三角按钮都可以打开虚拟机，但是此时虚拟机没有安装任何操作系统，因此没法打开，接下来我们就是要在刚刚新建的这个虚拟机中安装 Ubuntu 操作系统。

1.3 安装 Ubuntu 操作系统

1.3.1 获取 Ubuntu 系统

前面虚拟机已经创建成功了，相当于硬件已经准备好了，接下来就是要在虚拟机中安装

Ubuntu 系统了，首先肯定是获取到 Ubuntu 的系统镜像，Ubuntu 系统镜像肯定是在 Ubuntu 官网获取，下载地址为：https://www.ubuntu.com/download/desktop，如图 1.3.1.1 所示：

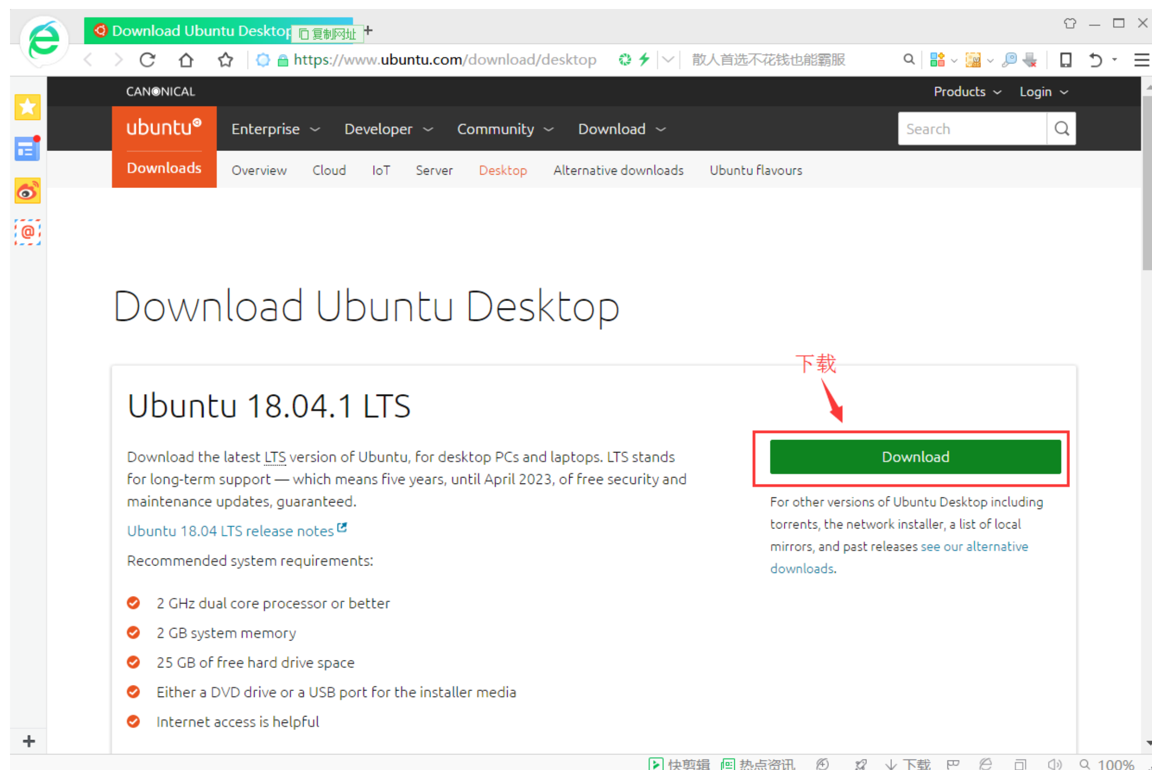


图 1.3.1.1 Ubuntu 最新版系统下载

从图 1.3.1.1 中可以看出，最新版本的 Ubuntu 系统是 18.04，但是在笔者编写本教程的时候 18.04 版刚出来没多久，怕不稳定，因此笔者实际使用的是 16.04 版本的 Ubuntu，后面所有的例程和教程均在 16.04 下完成，包括我们接下来安装的也是 16.04 版本的 Ubuntu。16.04 版本的 Ubuntu 下载地址为：http://releases.ubuntu.com/16.04/，下载"ubuntu-16.04.5-desktop-amd64.iso"这个版本，我已经下载下来放到了开发板光盘中，路径为：3、软件-> ubuntu-16.04.5-desktop-amd64.iso。

1.3.2 安装 Ubuntu 操作系统

Ubuntu 系统获取到以后就可以安装了，打开 VMware 软件，选择：虚拟机->设置，如图 1.3.2.1 所示：

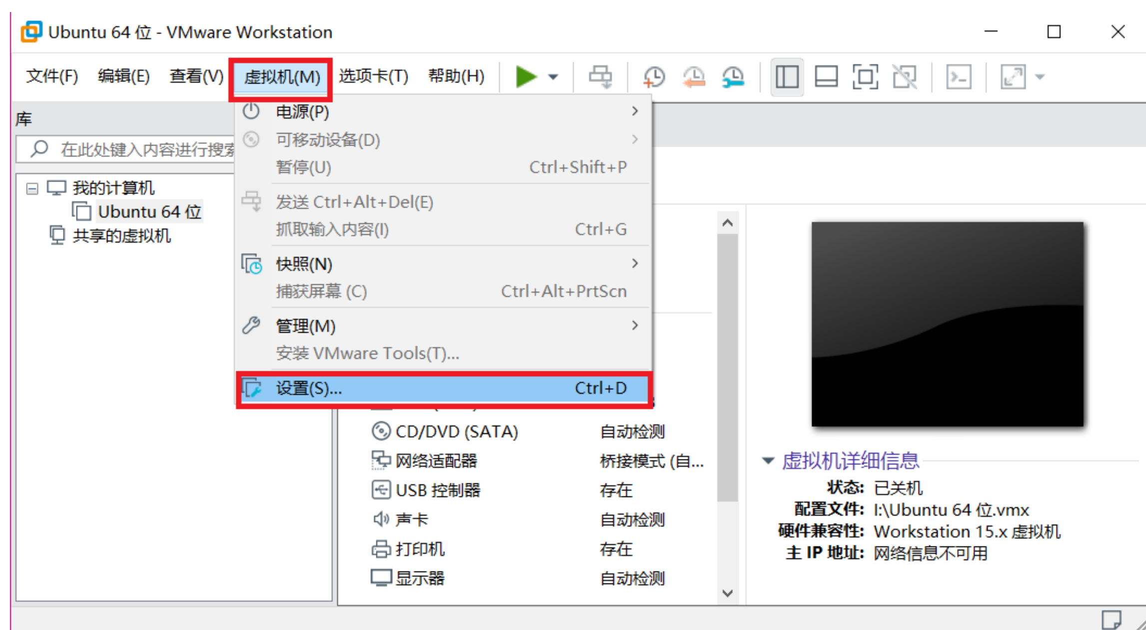


图 1.3.2.1 打开虚拟机设置对话框

打开以后的虚拟机设置对话框如图 1.3.2.2 所示：



图 1.3.2.2 虚拟机对话框

首先设置“USB 控制器”选项，默认 USB 控制器的 USB 兼容性为 USB2.0，这样当你使用 USB3.0 的设备的时候 Ubuntu 可能识别不出来，因此我们需要调整 USB 兼容性为 USB3.0，如图 1.3.2.3 所示：



图 1.3.2.3 USB 兼容性设置

设置要 USB 兼容性以后就开始安装 Ubuntu 系统了，选中虚拟机设置对话框中的“CD/DVD(SATA)”选项，然后在右侧选中“使用 ISO 映像文件”，如图 1.3.2.4 所示：

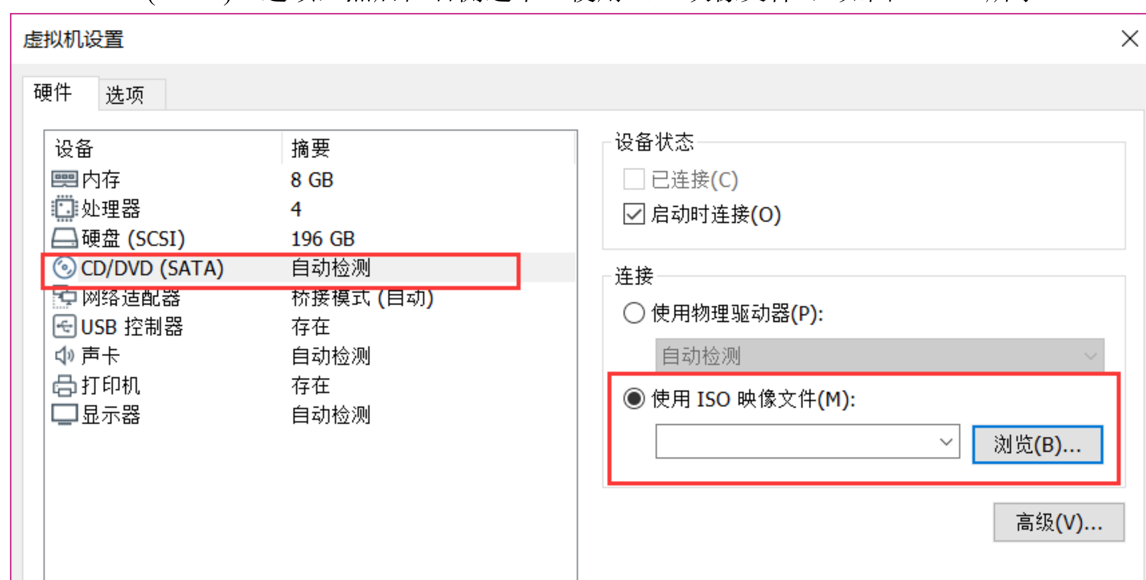


图 1.3.2.4 系统镜像设置

在图 1.3.2.4 中的“使用 ISO 映像文件”里面添加我们刚刚下载到的 Ubuntu 系统镜像，点击“浏览”按钮，选择 Ubuntu 系统镜像，完成以后如图 1.3.2.5 所示：

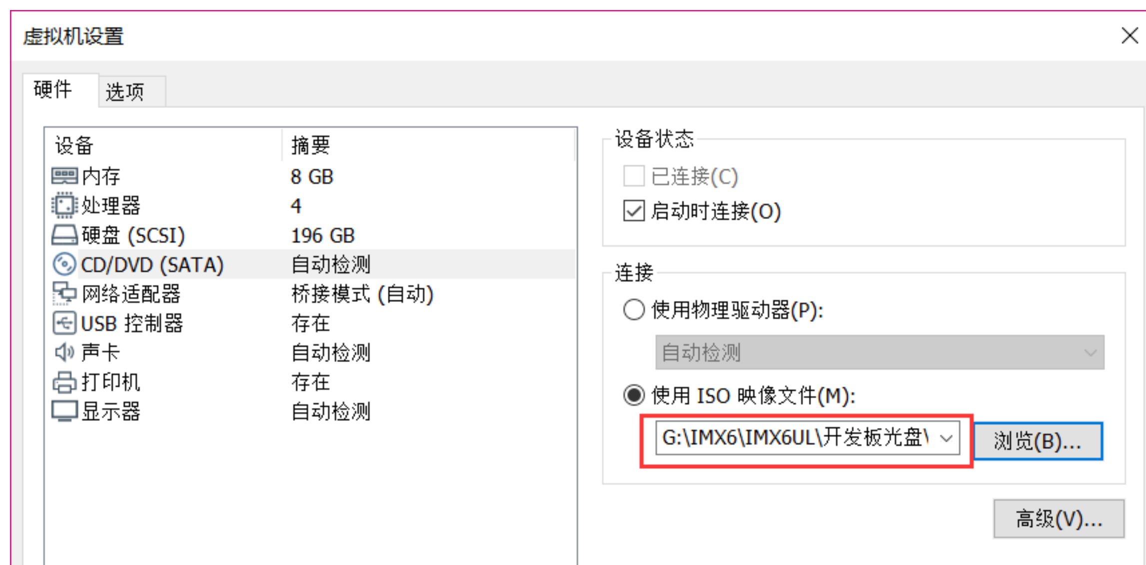


图 1.3.2.5 Ubuntu 镜像选择

设置好以后点击“确定”按钮退出，退出以后就可以打开虚拟机了，虚拟机就会自动的安装 Ubuntu 系统，如图 1.3.2.6 所示：

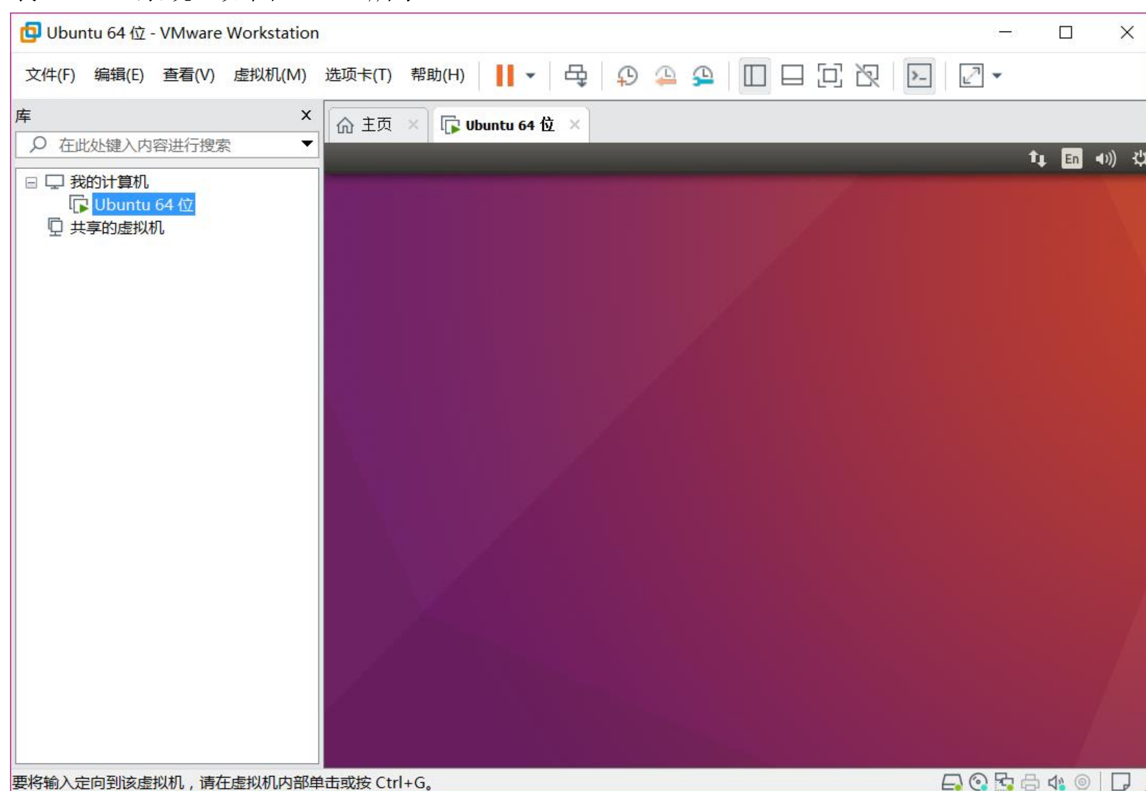


图 1.3.2.6 Ubuntu 安装开始

Ubuntu 开始安装以后首先是语言选择，如图 1.3.2.7 所示：



图 1.3.2.7 语言选择

Ubuntu 默认语言是英文，毫无疑问，我们要选择"中文(简体)"，选择好以后点击右侧的"安装 Ubuntu"按钮，进入安装过程。安装一开始会有 7 个配置步骤，第一配置如图 1.3.2.8 所示，让你选择是否安装 Ubuntu 时下载更新，以及是否为图形或者无线硬件安装其它第三方软件，我们不勾选这两个，否则安装过程很慢。

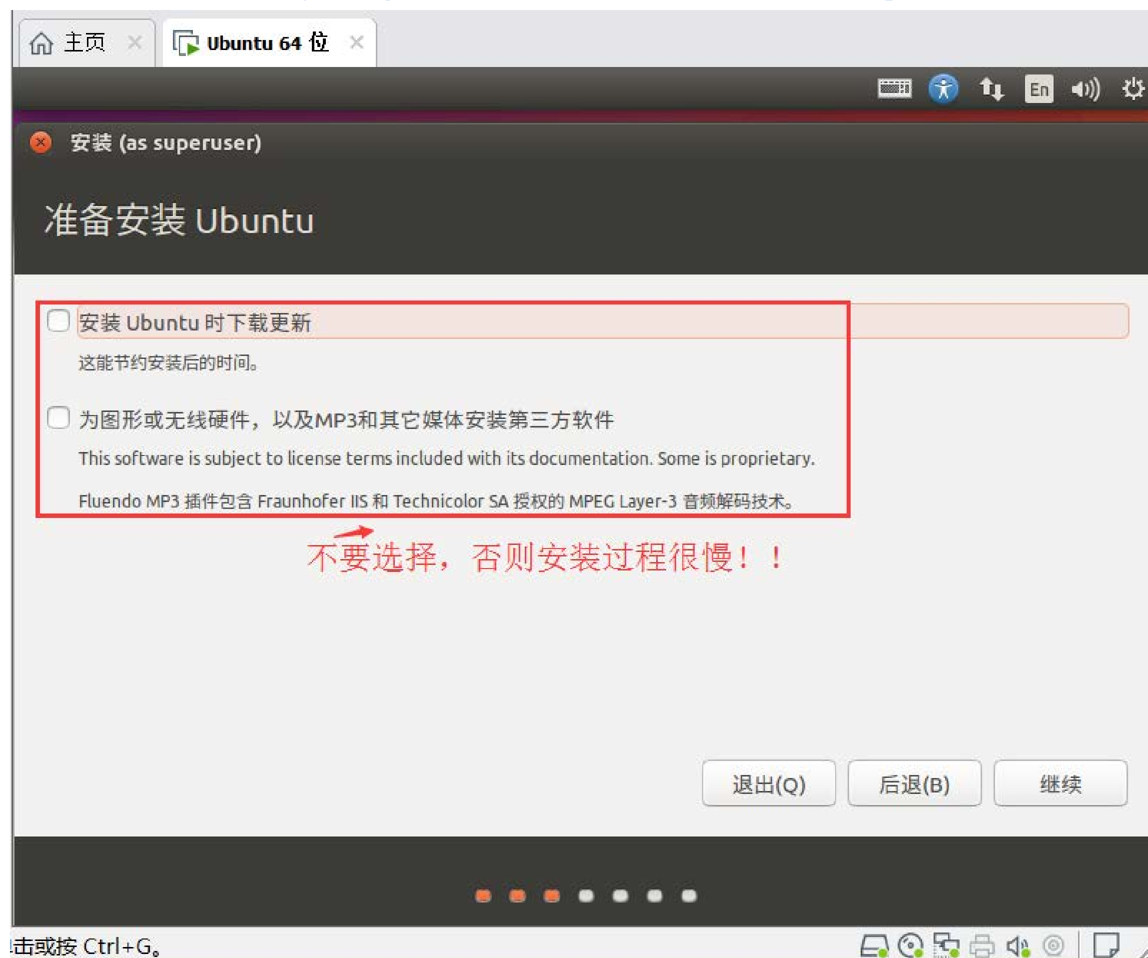


图 1.3.2.8 是否安装是下载更新

直接点击图 1.3.2.8 中的“继续”按钮，弹出安装类型，使用默认的“清除整个磁盘并安装 Ubuntu”，如图 1.3.2.9 所示：



图 1.3.2.9 安装类型选择

设置好安装类型以后点击“现在安装”按钮，会弹出“将改动写入磁盘吗？”对话框，点击“继续”即可，下一步会让你输入你在哪个位置，输入自己所在的城市即可，比如我在广州就输入广州，如图 1.3.2.10 所示：

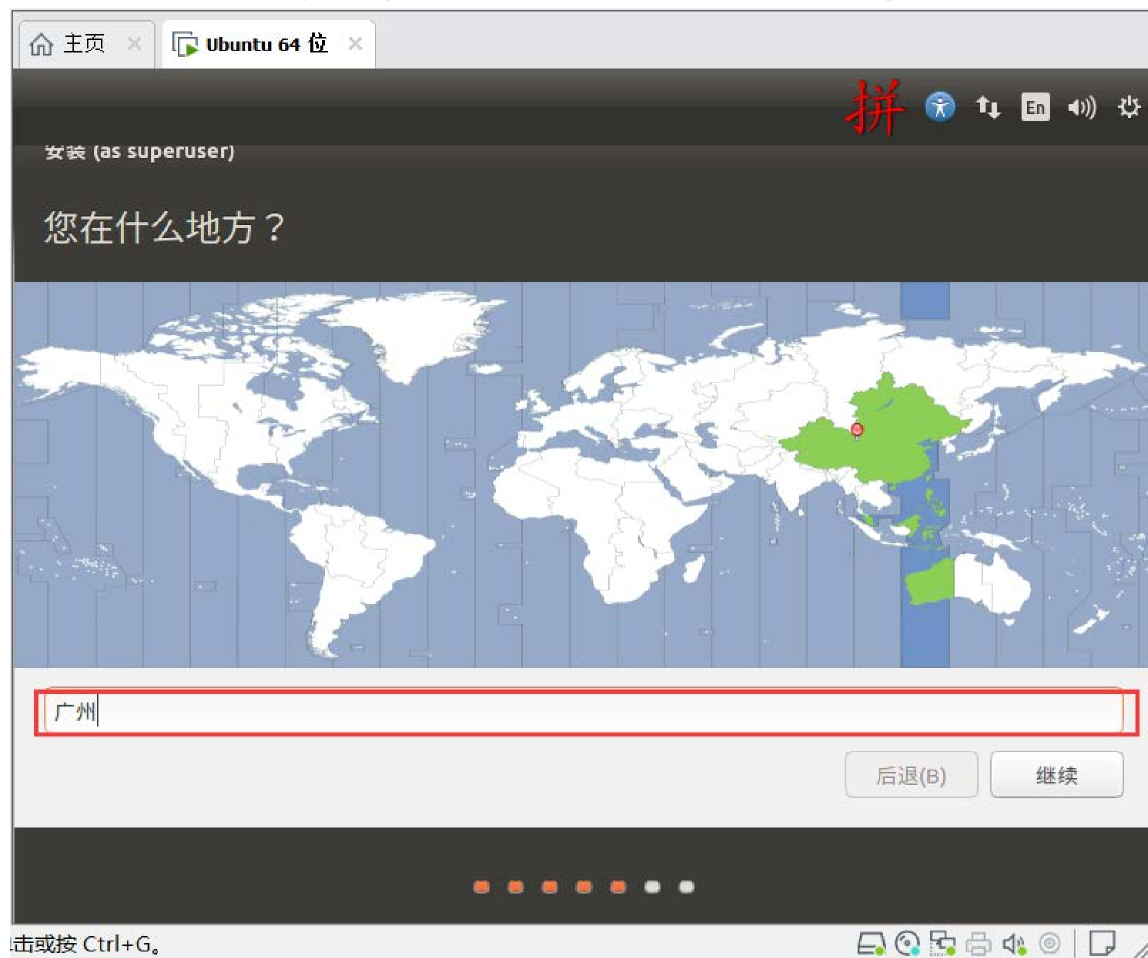


图 1.3.2.10 输入所在位置

Ubuntu 默认带了拼音输入法，切换到拼音输入法的方法如图 1.3.2.11 所示：

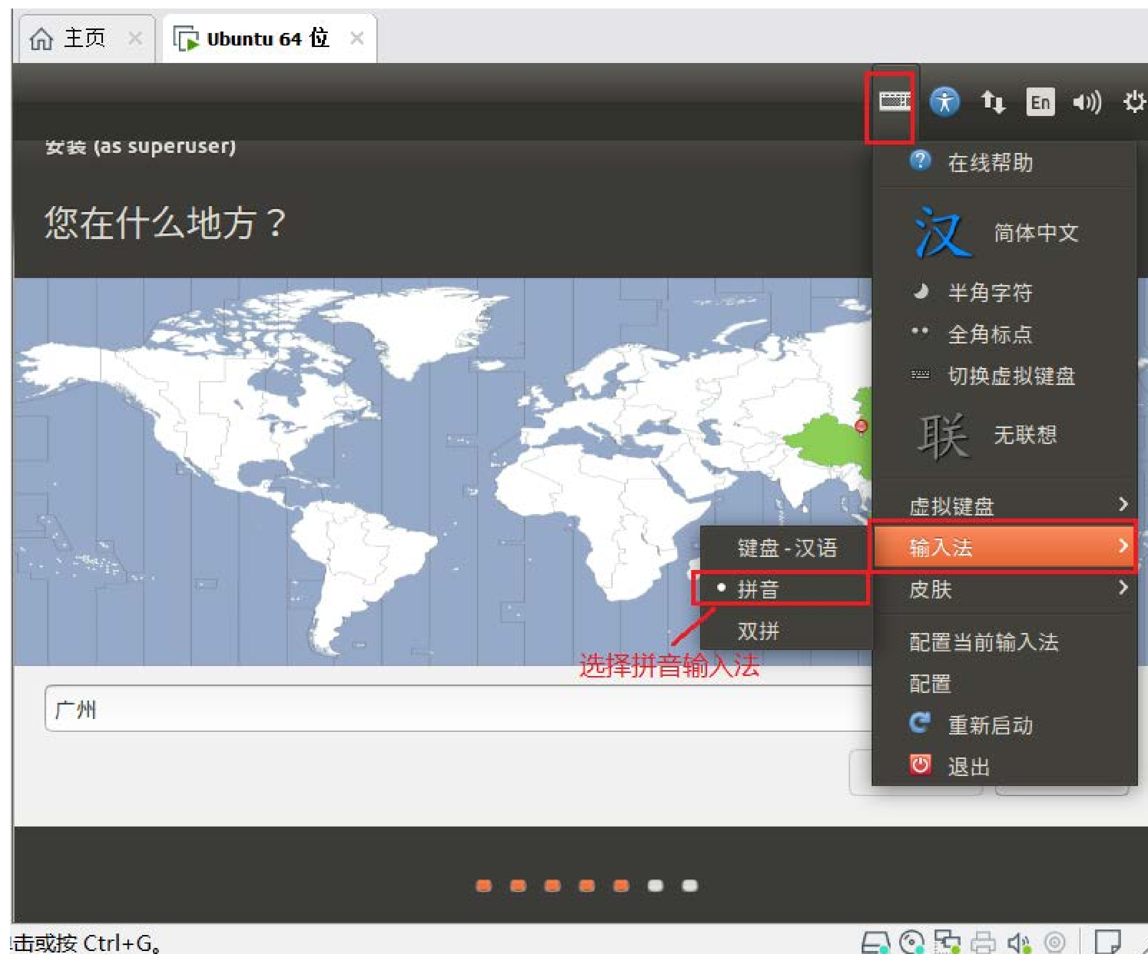


图 1.3.2.11 切换拼音输入法

输入地址以后点击“继续”按钮，会进入键盘布局设置界面，不需要做任何修改，点击“继续”按钮，进入下一步设置用户名和密码，自己设置自己的用户名和密码，比如我的设置如图 1.3.2.12 所示：



图 1.3.2.12 设置用户名和密码

设置好用户名和密码以后点击“继续”按钮，系统就会开始正式安装，如图 1.3.2.13 所示：



图 1.3.2.13 系统安装中

等待系统安装完成，安装过程中会下载一些文件，所以一定要保证电脑能够正常上网，如果不能正常上网的话可以点击右侧的"skip"按钮来跳过下载文件这个步骤，对于系统的安装没有任何影响，安装完成以后提示重启系统，如图 1.3.2.14 所示：

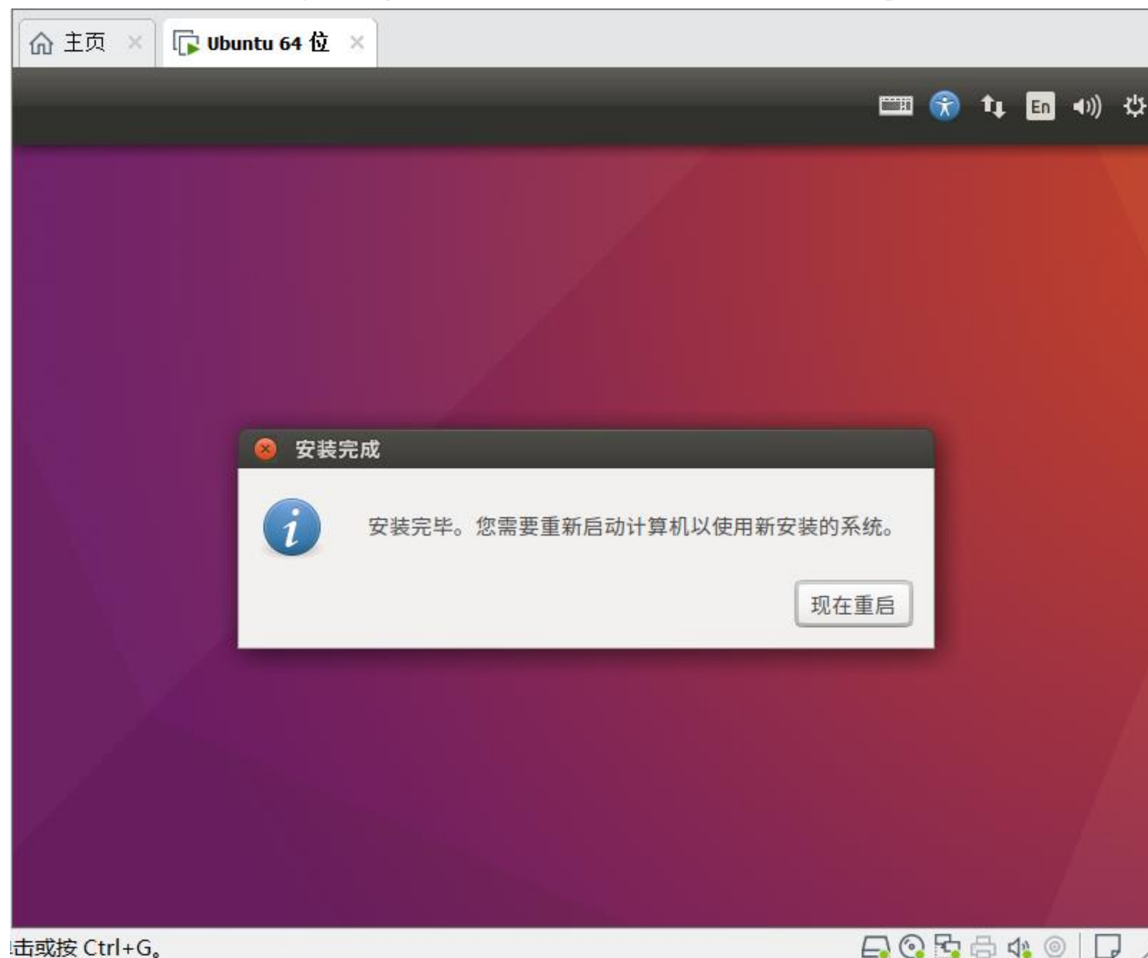


图 1.3.2.14 安装完成，重启系统

重启系统以后就会提示输入密码，如图 1.3.2.15 所示：

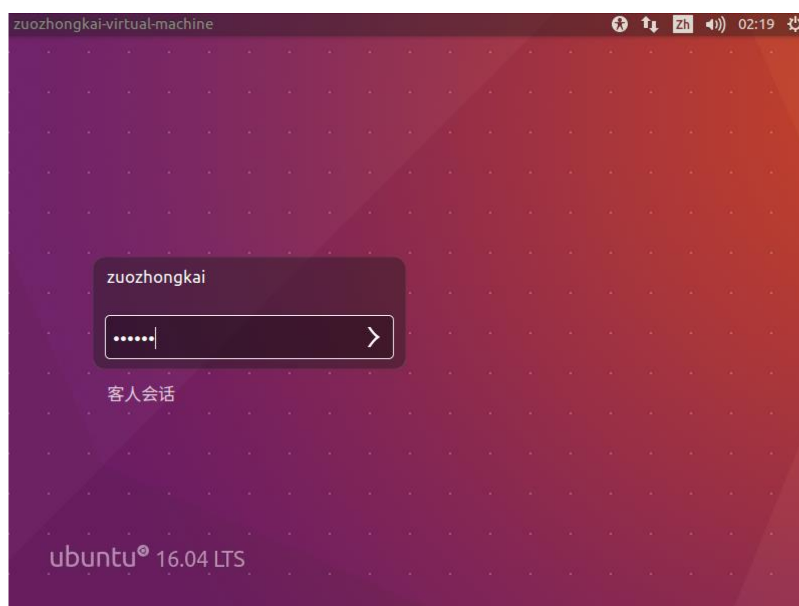


图 1.3.2.15 系统重启，输入密码

在图 1.3.2.15 中输入密码，点击键盘上的回车就会进入系统主界面，系统界面如图 1.3.2.16 所示：

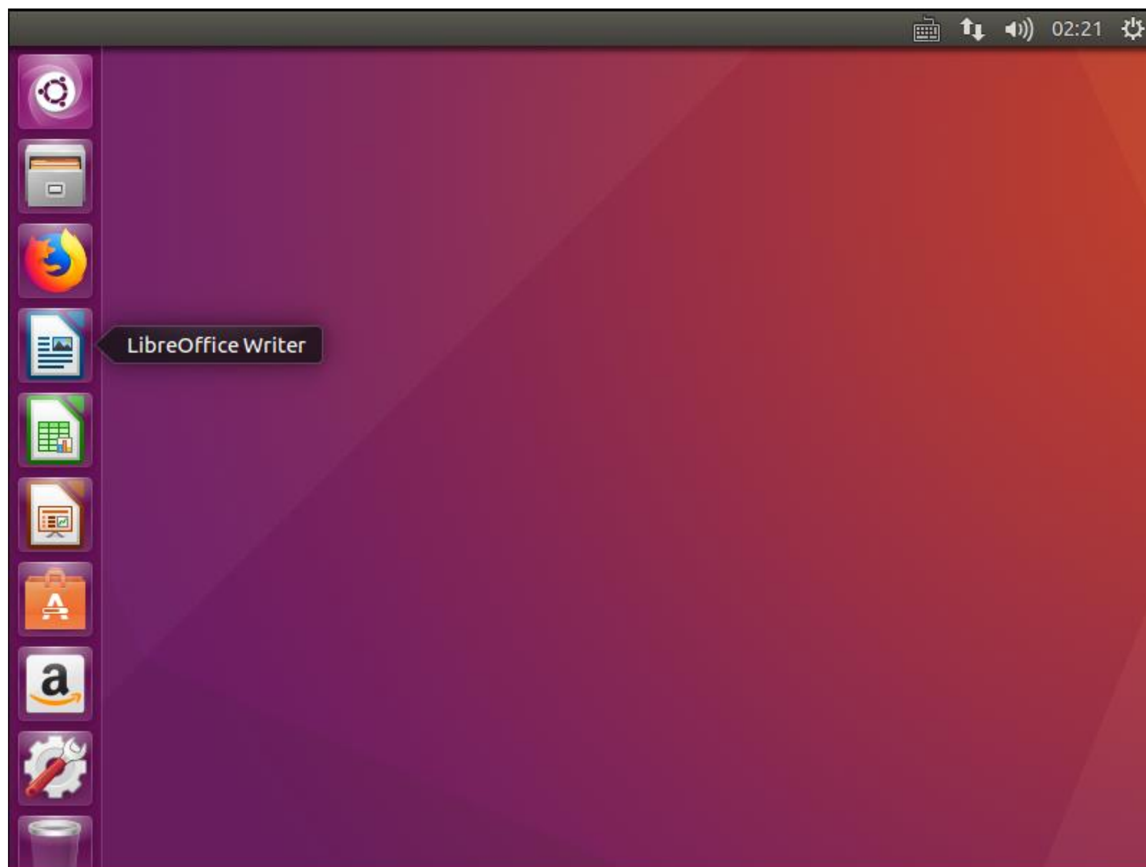


图 1.3.2.16 系统桌面

图 1.3.2.16 就是第一次进入系统的系统桌面，此时我们的系统镜像还在 CD/DVD 里面，我们要将它弹出，先关闭 Ubuntu 系统，点击系统桌面右上角的设置按钮，如图 1.3.2.17 所示：



图 1.3.2.17 关机

按照图 1.3.2.17 所示方式关机。

1.3.3 弹出系统镜像

和我们在真实电脑上安装系统一样，不管我们使用的光盘还是 U 盘安装系统，当系统安装成功以后都要弹出光盘或者拔出 U 盘，然后调整 BIOS 从硬盘启动，否则以后开机的话都会首

先从光盘或者 U 盘启动了，这样会进入系统安装界面。同理，我们在 VMware 中安装 Ubuntu 的时候是在 CD/DVD 中加载了 Ubuntu 系统镜像，现在系统安装成功了，因此也要把这个镜像从 CD/DVD 中弹出。

关闭 Ubuntu 操作系统，关重新打开 VMware，不要打开 Ubuntu 系统！打开 VMware 的虚拟机设置界面，然后选中“CD/DVD(SATA)”，右侧的“连接”选择“使用物理驱动器”，如图 1.3.3.1 所示。

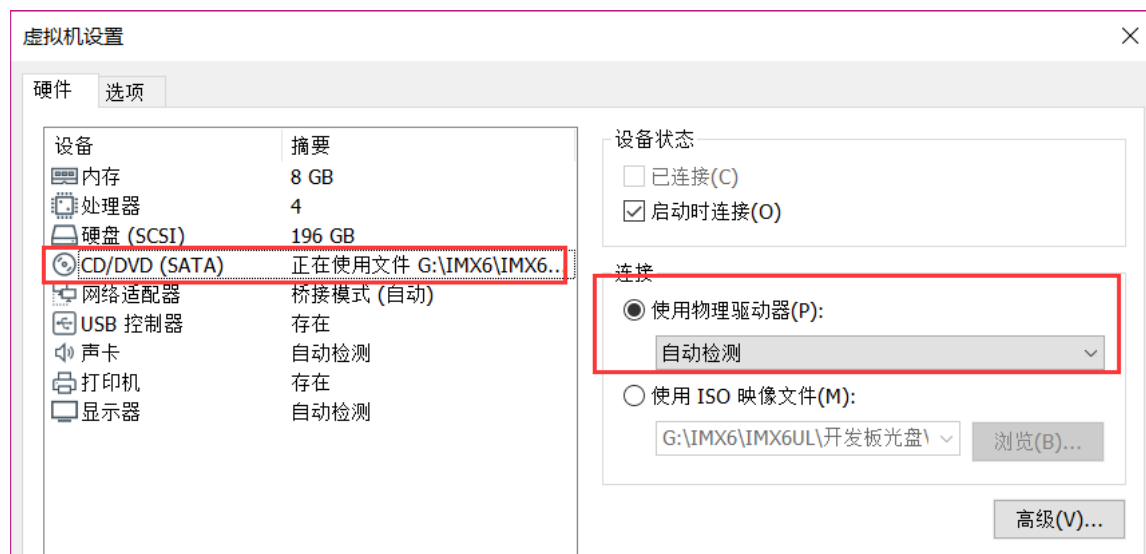


图 1.3.3.1 弹出 Ubuntu 系统镜像

设置好以后点击“确定”按钮，然后重新打开虚拟机，看看是否能够正常启动 Ubuntu，一般肯定能正常打开的。

至此，VMware 虚拟机以及 Ubuntu 系统安装成功，接下来我们就要学习如何是用 Ubuntu 了。

第二章 Ubuntu 系统入门

在上一章我们已经安装好虚拟机，并且在虚拟机中安装好了 Ubuntu 操作系统了，本章我们就来学习 Ubuntu 系统的基本使用，通过本章的学习为我们以后的开发做准备。Ubuntu 系统是和 Windows 一样的大型桌面操作系统，因此功能非常强大，不是一章就能介绍完的，因此本章叫做《Ubuntu 系统入门》。本章的主要目的是教会读者掌握后续嵌入式开发所需的 Ubuntu 基本技能，比如系统的基本设置、常用的 shell 命令、vim 编辑器的基本操作等等，如果想详细的学习 Ubuntu 操作系统请参考其它更为详实的书籍，本章参考了《Ubuntu Linux 从入门到精通》，这本书不厚，很适合用来作 Ubuntu 入门。

2.1 Ubuntu 系统初体验

2.1.1 Hello Ubuntu

上一章我们已经安装好了 Ubuntu 操作系统，我们再来回顾一下如何开机：

1、打开 VMware 虚拟机软件，打开以后如图 2.1.1.1 所示：

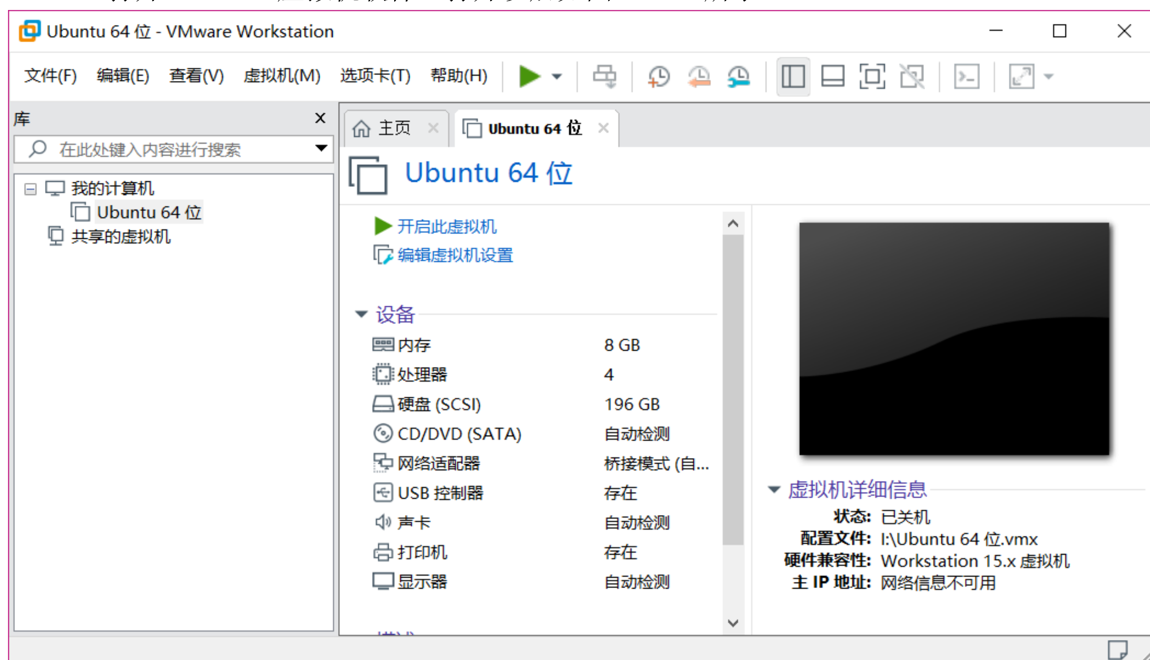


图 2.1.1.1 WMware 主界面

2、打开 VMware 上的开机按钮，打开方式如图 2.1.1.2 所示：



图 2.1.1.2 VMware 开机按钮

3、点击图 2.1.1.2 中两个开机按钮中的任意一个就会打开 Ubuntu 操作系统，首先进入图 2.1.1.3 所示的登陆界面，输入密码即可进入系统。

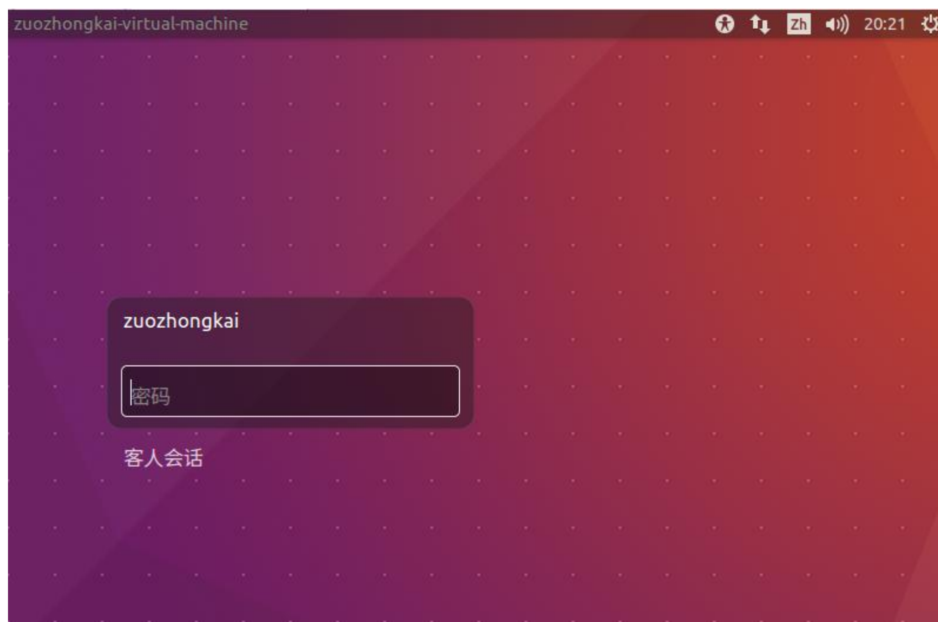


图 2.1.1.3 Ubuntu 登陆界面

在登陆界面输入密码，进入系统主界面，如图 2.1.1.4 所示：

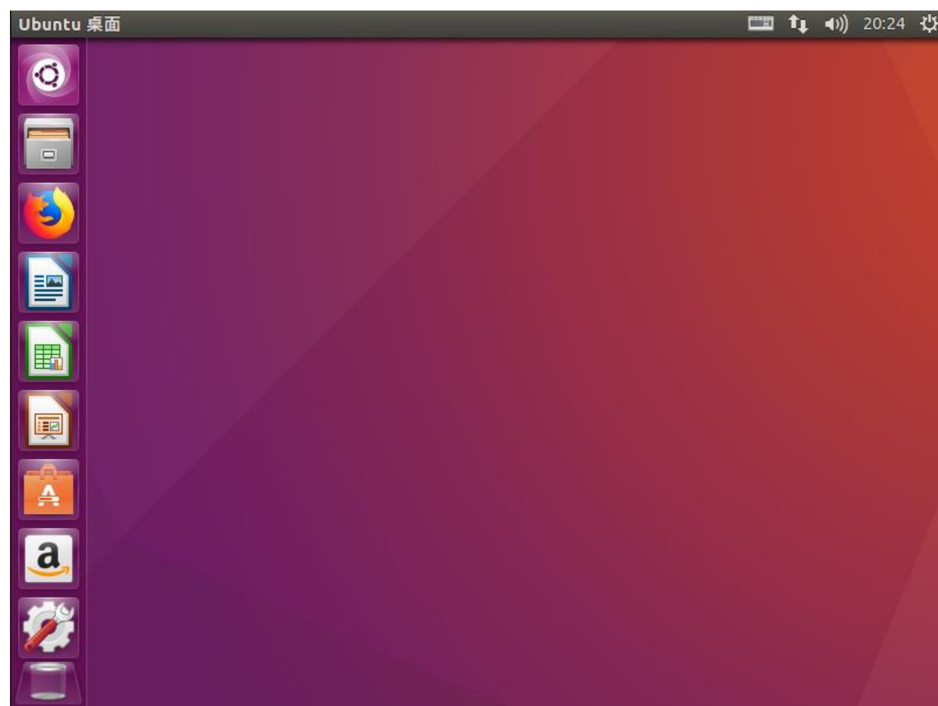


图 2.1.1.4 Ubuntu 主界面

进入主界面以后大家就可以看到和 Windows 基本一样，左侧有一列 APP，第一个是“搜索计算机”，第二个是文件浏览器，打开以后可以浏览 Ubuntu 系统中的文件，打开以后如图 2.1.1.5 所示：

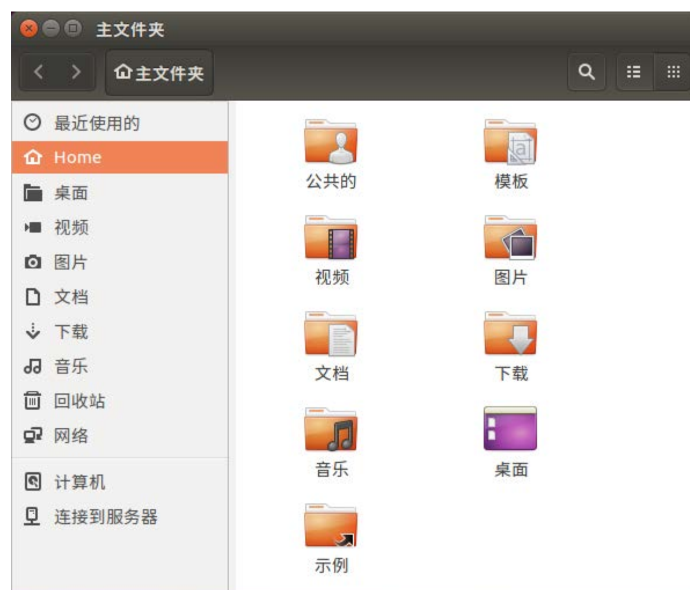


图 2.1.1.5 文件浏览

第三个是 firefox 浏览器，可以用来上网，比如我们登陆百度网站，如图 2.1.1.6 所示：

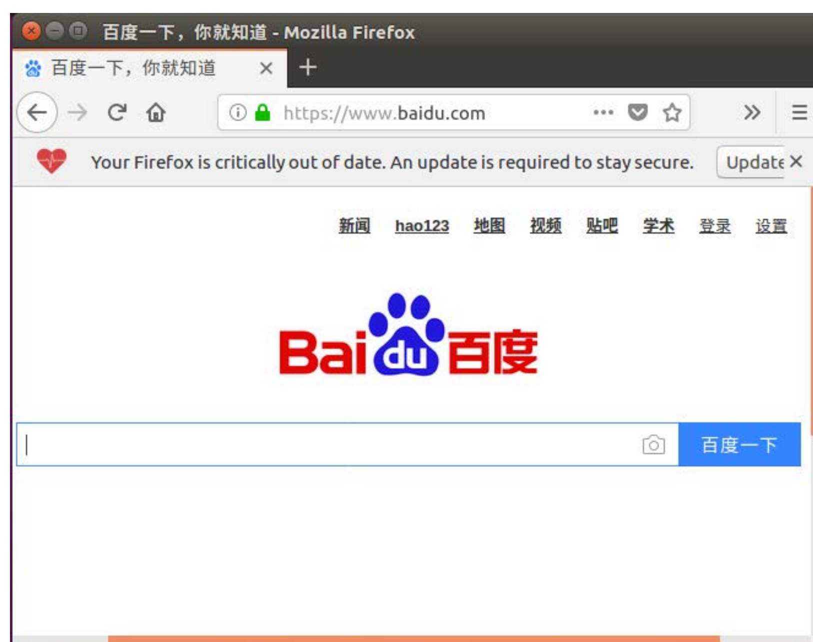


图 2.1.1.6 firefox 浏览器

这里还有其它一些 APP，大家可以自行打开看一下这些 APP 都是干啥的，这里就不一一详细的介绍了。

2.1.2 系统设置

我们会发现，Ubuntu 的默认桌面很小，这是因为 Ubuntu 默认分辨率是 800*600，因此我们首先要设置系统分辨率，调整到合适的大小。打开系统设置界面，打开方式如图 2.1.2.1 所示：

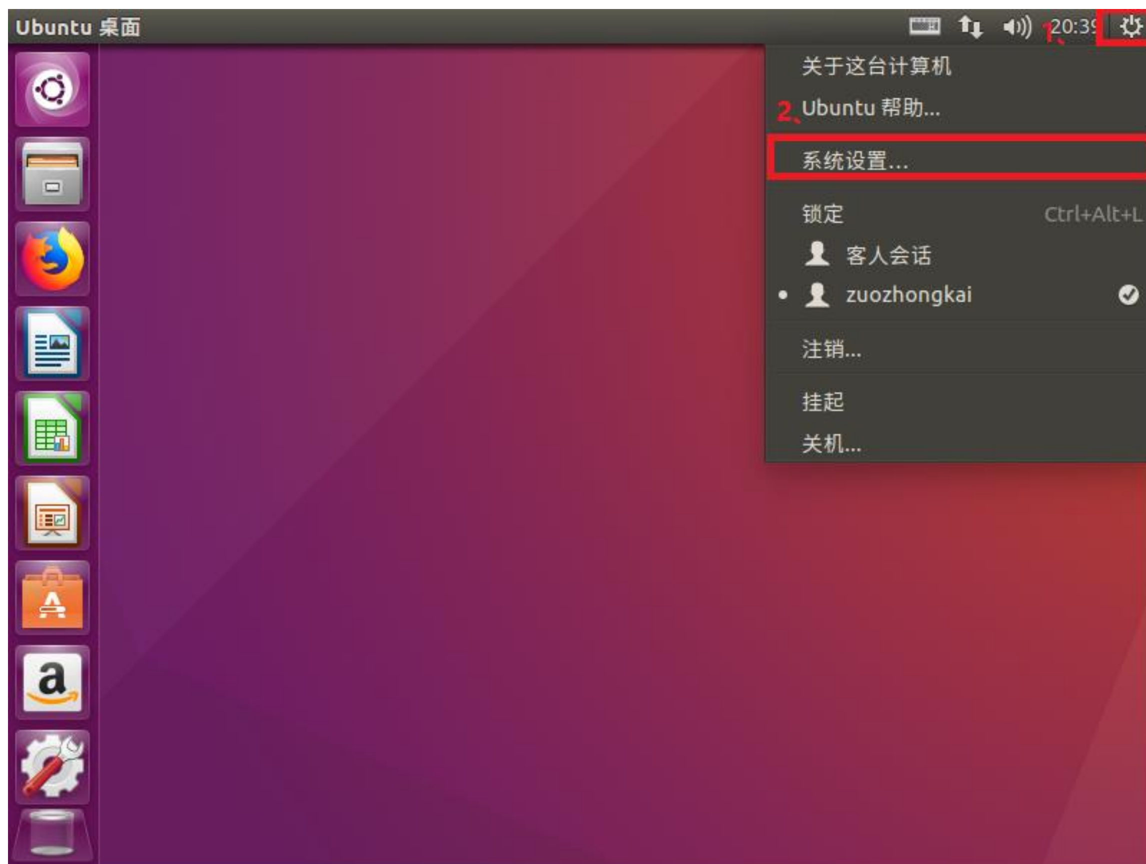


图 2.1.2.1 打开系统设置

打开以后的系统设置界面如图 2.1.2.2 所示：



图 2.1.2.2 系统设置界面

系统设置界面可以完成系统的大部分设置，我们找到“显示”设置并打开，打开以后如图 2.1.2.3 所示：



图 2.1.2.3 显示设置界面

从图 2.1.2.3 中可以看出，系统默认分辨率是 800X600，现在的电脑分辨率最少都是 1920X1080 了，因此我们可以调整这个分辨率至合适的大小，比如我设置为 1440x900 分辨率，设置好以后点击“应用”按钮，这里要注意，由于分辨率太小了，导致“应用”按钮就只露出了很少一部分，如图 2.1.2.4 所示：



图 2.1.2.4 调整屏幕分辨率

设置好分辨率以后 Ubuntu 的主界面就大了，看起来也舒服了。通过设置系统分辨率这个例子，我们就知道了如何设置 Ubuntu 系统，如果有需要设置其它东西的话都可以到系统设置里面去进行，这里就不一一详细的介绍了。

2.1.3 系统注销与关机

当我们不使用 Ubuntu 系统以后就需要将其关机，就和我们使用 Windows 系统一样，千万不要通过直接退出 VMware 软件来关机！！关机很简单，在主界面，点击右上角的齿轮图标，然后选择关机选项，如图 2.1.3.1 所示：



图 2.1.3.1 关机

在图 2.1.3.1 中可以看到有三个选项：注销，挂起和关机，这个和 Windows 下是一样的，你如果想要注销就点击“注销”按钮，想要关机就点击“关机”按钮，以关机为例，点击关机以后会弹出图 2.1.3.2 所示关机确认界面，在确认界面上可以选择是“重启”还是“关机”。



图 2.1.3.2 关机确认界面

在图 2.1.3.2 中，左边的按钮为重启图标，点击以后系统重启，右边的按钮为关机按钮，点击以后就会关闭 Ubuntu 系统。

2.1.4 中文输入测试

我们是中国人，平时用的做多的肯定是中文，那么 Ubuntu 下中文输入是否和 Windows 一样呢？如何在 Ubuntu 下使用中文输入法。我们在安装 Ubuntu 系统的时候就已经使用过中文输

入法了，就是选择我们所在地的时候。本节我们就以创建一个文本为例，介绍如何在 Ubuntu 中使用中文输入法。

在桌面上点击鼠标右键，然后选择新建文档->空白文档，如图 2.1.4.1 所示：



图 2.1.4.1 新建空白文档

文档名字使用默认名字：无标题文档，如图 2.1.4.2 所示：

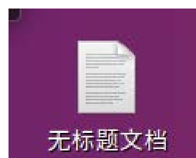


图 2.1.4.2 新建的无标题文档

双击打开文档，打开以后如图 2.1.4.3 所示：



图 2.1.4.3 打开文档

打开文档以后，我们可以尝试在里面输入一些英文和数字，输入英文和数字是没有任何问题的，输入中文的话需要切换到 Ubuntu 自带的拼音输入法，有两种方式切换，一种是使用快捷键:Windows+空格键，一种是使用鼠标点击设置输入法，如图 2.1.4.4 所示：



图 2.1.4.4 切换拼音输入法

这两种方法都可以切换输入法，切换到拼音输入法以后就可以输入中文了，如图 2.1.4.5 所示：



图 2.1.4.5 输入中文文本

大家会发现 Ubuntu 下的拼音输入法使用起来跟 Windows 下的输入法差距太大了，没有 Windows 下的输入法好用，没办法，谁让桌面端 Linux 用的少呢，所以也就没有啥公司开发 Linux 下的输入法。

通过上面几个小节中对 Ubuntu 的基本操作来看，基本和 Windows 下的操作差不多，我们真正要使用 Ubuntu 的不是通过图形界面操作，而是通过命令行操作的。这也是我们接下来着重要讲的：Ubuntu(Linux)终端操作，会涉及到很多命令，但是常用的命令就那几十个，不需要刻

意的去背，使用习惯了就自然记住了。不要看到要记命令就觉得可怕。根据 2080 原则，80%情况下只使用那 20%的命令，实际情况会更少，常用的可能就那 5%~10%的命令。

2.2 Ubuntu 终端操作

本节就是我们学习 Ubuntu 操作系统的重点了，终端操作，也就是俗称的“敲命令”，不管是哪个版本的 Linux 发行版系统，它都会提供终端操作，Linux 下的终端操作类似与 Windows 下的 DOS 操作。要使用终端首先肯定是要打开终端，在主界面上点击鼠标右键，然后选择打开终端，如图 2.2.1 所示：



图 2.2.1 打开终端

打开终端以后如图 2.2.2 所示：

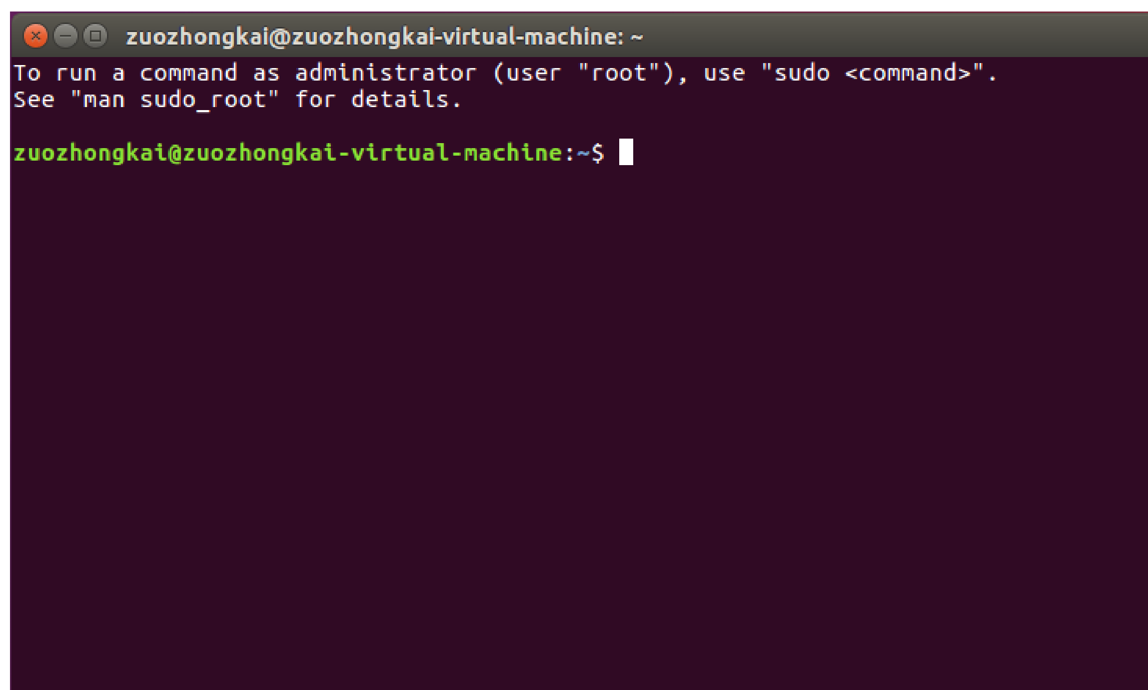


图 2.2.2 终端界面

我们就是在图 2.2.2 所示界面上输入命令的，终端默认会有类似下面一行所示的一串提示符:

```
zuozhongkai@zuozhongkai-virtual-machine: ~$
```

上述字符串中，@前面的“zuozhongkai”是当前的用户名字，@后面的 zuozhongkai-virtual-

machine 是我的机器名字。最后面的符号“$”表示当前用户是普通用户，我们可以在提示符后面输入命令，比如输入命令“ls”，命令“ls”是打印出当前所在目录中所有文件和文件夹，如图 2.2.3 所示：

```
zuozhongkai@ubuntu: ~
zuozhongkai@ubuntu:~$ ls
examples.desktop  tmp  公共的  模板  视频  图片  文档  下载  音乐  桌面
zuozhongkai@ubuntu:~$
```

图 2.2.3 ls 命令

在图 2.2.3 中我们输入了“ls”这个命令，然后打印出了当前目录下的所有文件和文件夹，后面我们学习命令的时候就是在终端中输入相应命令的。

2.3 Shell 操作

2.3.1 Shell 简介

学习 linux 的时候会频繁的看到 Shell 这个词语？那么什么是 Shell 呢？网上搜索一下，各种专业的解释一堆，但是对于第一次接触 Linux 的人来说这些专业的词语只会让人更晕。简单的说 Shell 就是敲命令。国内把 Linux 下通过命令行输入命令叫做“敲命令”，国外人玩的比较洋气，人家叫做“Shell”。因此以后看到 Shell 这个词语第一反应就是在终端中敲命令，将多个 Shell 命令按照一定的格式放到一个文本中，那么这个文本就叫做 Shell 脚本。

严格意义上来讲，Shell 是一个应用程序，它负责接收用户输入的命令，然后根据命令做出相应的动作，Shell 负责将应用层或者用户输入的命令传递给系统内核，由操作系统内核来完成相应的工作，然后将结果反馈给应用层或者用户。

2.3.2 Shell 基本操作

前面我们说 Shell 就是“敲命令”，那么既然是命令，那肯定是有格式的，Shell 命令的格式如下：

```
command     -options      [argument]
```

command: Shell 命令名称。

options：选项，同一种命令可能有不同的选项，不同的选项其实现的功能不同。

argument：Shell 命令是可以带参数的，也可以不带参数运行。

同样以命令“ls”为例，下面“ls”命令的三种不同格式其结果也不同：

```
ls
ls -l
ls /usr
```

这三种命令的运行结果如图 2.3.2.1 所示：

```
zuozhongkai@ubuntu: ~
zuozhongkai@ubuntu:~$ ls
examples.desktop  tmp  公共的  模板  视频  图片  文档  下载  音乐  桌面
zuozhongkai@ubuntu:~$ ls -l
总用量 48
-rw-r--r-- 1 zuozhongkai zuozhongkai 8980 12月  18 02:08 examples.desktop
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  19 00:29 tmp
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 公共的
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 模板
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 视频
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 图片
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 文档
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 下载
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 音乐
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  19 00:25 桌面
zuozhongkai@ubuntu:~$ ls /usr
bin  games  include  lib  local  locale  sbin  share  src
zuozhongkai@ubuntu:~$
```

图 2.3.2.1 ls 命令

在图 2.3.2.1 中“ls”命令用来打印出当前目录下的所有文件和文件夹，而“ls -l”同样是打印出当前目录下的所有文件和文件夹，但是此命令会列出所有文件和文件夹的详细信息，比如文件大小、拥有者、创建日期等等。最有一个“ls /usr”是用来打印出目录“/usr”下的所有文件和文件夹。

Shell 命令是支持自动补全功能的，因为 Shell 命令非常多，如果不作自动补全的话就需要用户去记忆这些命令的全部字母。使用自动补全功能以后我们只需要输入命令的前面一部分字母，然后按下 TAB 键，如果只有一个命令匹配的话就会自动补全这个命令剩下的字母。如果有多个命令匹配的话系统就会发出报警声音，此时在按下一次 TAB 键就会列出所有匹配的命令，比如我们输入字母“if”，然后按下 TAB 键，结果如图 2.3.2.2 所示：

```
zuozhongkai@ubuntu:~$ if
if        ifconfig  ifdown    ifquery   ifup
```

图 2.3.2.2 “if”开始的命令

从图 2.3.2.2 可以看出，以“if”开头的命令有 5 个，我们以“ifconfig”为例，此命令是用来查看网卡信息的，我们重新输入“ifc”然后在按一下 TAB 键，就会自动补全出“ifconfig”命令，因为以“ifc”开头的命令只有一个，结果如图 2.3.2.3 所示：

```
zuozhongkai@ubuntu:~$ ifconfig
ens33     Link encap:以太网  硬件地址 00:0c:29:96:89:d6
          inet 地址:192.168.31.235  广播:192.168.31.255  掩码:255.255.255.0
          inet6 地址: fe80::e92a:d882:e1b:7402/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:14268 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:1571 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:5861098 (5.8 MB)  发送字节:109785 (109.7 KB)

lo        Link encap:本地环回
          inet 地址:127.0.0.1  掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  跃点数:1
          接收数据包:247 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:247 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:19425 (19.4 KB)  发送字节:19425 (19.4 KB)
```

图 2.3.2.3 ifconfig 命令结果

2.2.4 常用 Shell 命令

我们做嵌入式开发用的最多就是 Shell 命令，Shell 命令是所有的 Linux 系统发行版所通用的，并不是说我在 Ubuntu 下学会了 Shell 命令，换另外一个 Linux 发行版操作系统以后就没用了(不同的发行版 Linux 系统可能会自定义一些命令)。本节我们先来介绍一些 Shell 下常用的命令：

1、目录信息查看命令 ls

文件浏览是最基本的操作了，Shell 下文件浏览命令为 ls，格式如下：

```
ls    [选项]    [路径]
```

ls 命令主要用于显示指定目录下的内容，列出指定目录下包含的所有的文件以及子目录，它的主要参数有：

-a 显示所有的文件以及子目录，包括以“.”开头的隐藏文件。

-l 显示文件的详细信息，比如文件的形态、权限、所有者、大小等信息。

-t 将文件按照创建时间排序列出。

-A 和-a 一样，但是不列出“.”(当前目录)和“..”(父目录)。

-R 递归列出所有文件，包括子目录中的文件。

Shell 命令里面的参数是可以组合在一起用的，比如组合“-al”就是显示所有文件的详细信息，包括以“.”开头的隐藏文件，ls 命令使用如图 2.2.4.1 所示：

```
zuozhongkai@ubuntu:~/tmp$ ls
a  b  c
zuozhongkai@ubuntu:~/tmp$ ls -a
.  ..  a  b  c
zuozhongkai@ubuntu:~/tmp$ ls -al
总用量 8
drwxrwxr-x  2 zuozhongkai zuozhongkai 4096 12月  19 21:41 .
drwxr-xr-x 20 zuozhongkai zuozhongkai 4096 12月  19 20:31 ..
-rw-rw-r--  1 zuozhongkai zuozhongkai    0 12月  19 21:41 a
-rw-rw-r--  1 zuozhongkai zuozhongkai    0 12月  19 21:41 b
-rw-rw-r--  1 zuozhongkai zuozhongkai    0 12月  19 21:41 c
```

图 2.2.4.1 ls 命令演示

注意图 2.2.4.1 中 tmp 文件夹是我为了演示方便，自己创建的，里面的文件 a，b 和 c 也是我创建的，关于文件夹和文件的创建后面会详细的讲解。

2、目录切换命令 cd

要想在 Shell 中切换到其它的目录，使用的命令是 cd，命令格式如下：

```
cd    [路径]
```

路径就是我们要进入的目录路径，比如下面所示操作：

```
cd  /        //进入到根目录“/”下，Linux 系统的根目录为“/”，
cd  /usr     //进入到目录“/usr”里面。
cd  ..       //进入到上一级目录。
cd  ~        //切换到当前用户主目录
```

比如我们要进入到目录“/usr”下去，并且查看“/usr”下有什么文件，操作如图 2.2.4.2 所示：

```
zuozhongkai@ubuntu:~$ cd /usr
zuozhongkai@ubuntu:/usr$ ls
bin  games  include  lib  local  locale  sbin  share  src
```

图 2.2.4.2 cd 命令演示

在图 2.2.4.2 中，我们先使用命令“cd /usr”进入到“/usr”目录下，然后使用“ls”命令显示“/usr”目录下的所有文件。仔细观察图 2.2.4.2 可以看到，当我们切换到其它目录以后在符号“$”前面就会以蓝色的字体显示出当前目录名字，如图 2.2.4.3 所示：

```
zuozhongkai@ubuntu:/usr$ ls
bin  games  include  lib  local  locale  sbin  share  src
```

图 2.2.4.3 目录路径显示

3、当前路径显示命令 pwd

pwd 命令用来显示当前工作目录的绝对路径，不需要任何的参数，使用如图 2.2.4.4 所示：

```
zuozhongkai@ubuntu:~$ pwd
/home/zuozhongkai
```

图 2.2.4.4 pwd 命令

4、系统信息查看命令 uname

要查看当前系统信息，可以使用命令 uname，命令格式如下：

```
uname    [选项]
```

可选的选项参数如下：

-r 列出当前系统的具体内核版本号。

-s 列出系统内核名称。

-o 列出系统信息。

使用如图 2.2.4.5 所示：

```
zuozhongkai@ubuntu:~$ uname
Linux
zuozhongkai@ubuntu:~$ uname -r
4.15.0-29-generic
zuozhongkai@ubuntu:~$ uname -s
Linux
zuozhongkai@ubuntu:~$ uname -o
GNU/Linux
```

图 2.2.4.5 uanme 命令操作

5、清屏命令 clear

clear 命令用于清除终端上的所有内容，只留下一行提示符。

6、切换用户执行身份命令 sudo

Ubuntu(Linux)是一个允许多用户的操作系统，其中权限最大的就是超级用户 root，有时候我们执行一些操作的时候是需要用 root 用户身份才能执行，比如安装软件。通过 sudo 命令可以使我们暂时将身份切换到 root 用户。当使用 sudo 命令的时候是需要输入密码的，这里要注意输入密码的时候是没有任何提示的！命令格式如下：

```
sudo    [选项]    [命令]
```

选项主要参数如下：

-h 显示帮助信息。

-l　列出当前用户可执行与不可执行的命令

-p　改变询问密码的提示符。

假如我们现在要创建一个新的用户 test，创建新用户的命令为“adduser”，创建新用户的权限只有 root 用户才有，我们在装系统的时候创建的那个用户是没有这个权限的，比如我的“zuozhongkai”用户。所以创建新用户的话需要使用“sudo”命令以 root 用户执行“adduser”这个命令，如图 2.2.4.6 所示：

```
zuozhongkai@ubuntu:~$ adduser test
adduser: 只有 root 才能将用户或组添加到系统。
zuozhongkai@ubuntu:~$ sudo adduser test
正在添加用户"test"...
正在添加新组"test" (1001)...
正在添加新用户"test" (1001) 到组"test"...
创建主目录"/home/test"...
正在从"/etc/skel"复制文件...
输入新的 UNIX 密码：
重新输入新的 UNIX 密码：
passwd: 已成功更新密码
正在改变 test 的用户信息
请输入新值，或直接敲回车键以使用默认值
        全名 []:
        房间号码 []:
        工作电话 []:
        家庭电话 []:
        其它 []:
这些信息是否正确？ [Y/n] y
```

图 2.2.4.6 sudo 命令演示

在图 2.2.4.6 中，我们一开始直接使用“adduser　test”命令添加用户的时候提示我们“adduser：只有 root 才能将用户或组添加到系统。”所以我们要在前面加上“sudo”命令，表示以 root 用户执行 adduser 操作。

7、添加用户命令 adduser

在讲解 sudo 命令的时候我们已经用过命令“adduser”，此命令需要 root 身份去运行。命令格式如下：

```
adduser  [参数]  [用户名]
```

常用的参数如下：

-system　添加一个系统用户

-home DIR　DIR 表示用户的主目录路径

-uid ID　ID 表示用户的 uid。

-ingroup GRP　表示用户所属的组名。

adduser 的使用我们前面已经演示过了，大家可以试着再添加一个用户。

8、删除用户命令 deluser

前面讲了添加用户的命令，那肯定也有删除用户的命令，删除用户使用命令“deluser”，命令参数如下：

```
deluser   [参数]  [用户名]
```

主要参数有：

-system　当用户是一个系统用户的时候才能删除。

-remove-home　删除用户的主目录

-remove-all-files　删除与用户有关的所有文件。

-backup　　　　备份用户信息

同样的，命令“deluser”也要使用“sudo”来以 root 用户运行，以删除我们前面创建的用户 test 为例，deluser 使用如图 2.2.4.7 所示：

```
zuozhongkai@ubuntu:~$ sudo deluser --remove-all-files test
正在寻找要备份或删除的文件...
/usr/sbin/deluser: 无法处理特殊文件 /sys/kernel/security/apparmor/.null
/usr/sbin/deluser: 无法处理特殊文件 /run/udev/static_node-tags/uaccess/snd\x2fseq
/usr/sbin/deluser: 无法处理特殊文件 /run/udev/static_node-tags/uaccess/snd\x2ftimer
/usr/sbin/deluser: 无法处理特殊文件 /dev/vcsa7
/usr/sbin/deluser: 无法处理特殊文件 /dev/vcs7

..................................
/usr/sbin/deluser: 无法处理特殊文件 /lib/systemd/system/single.service
/usr/sbin/deluser: 无法处理特殊文件 /lib/systemd/system/cryptdisks.service
/usr/sbin/deluser: 无法处理特殊文件 /lib/systemd/system/cryptdisks-early.service
正在删除文件...
正在删除用户 'test'...
警告：组"test"没有其他成员了。
完成。
```

图 2.2.4.7 命令 deluser 演示

9、切换用户命令 su

前面在讲解命令“sudo”的时候说过，“sudo”是以 root 用户身份执行一个命令，并没有更改当前的用户身份，所有需要 root 身份执行的命令都必须在前面加上“sudo”。命令“su”可以直接将当前用户切换为 root 用户，切换到 root 用户以后就可以尽情的进行任何操作了！因为你已经获得了系统最高权限，在 root 用户下，所有的命令都可以无障碍执行，不需要在前面加上“sudo”，“su”命令格式如下：

```
su [选项] [用户名]
```

常用选项参数如下：

-c –command　　执行指定的命令，执行完毕以后回复原用户身份。

-login　　改变用户身份，同时改变工作目录和 PATH 环境变量。

-m　　改变用户身份的时候不改变环境变量

-h　　显示帮助信息

以切换到 root 用户为例，使用如图 2.2.4.8 所示：

```
zuozhongkai@ubuntu:~$ sudo su
[sudo] zuozhongkai 的密码：
root@ubuntu:/home/zuozhongkai#
```

图 2.2.4.8 su 命令演示

在图 2.2.4.8 中，先使用命令“sudo su”切换到 root 用户，su 命令不写明用户名的话默认切换到 root 用户。然后输入密码，密码正确的话就会切换到 root 用户，可以看到切换到 root 用户以后提示符的“@”符号前面的用户名变成了“root”，表示当前的用户是 root 用户。并且以“#”结束。

注意!! 由于 root 用户权限太大，稍微不注意就可能删除掉系统文件，导致系统奔溃，因此强烈建议大家，不要以 root 用户运行 Ubuntu。当要用到 root 身份执行某些命令的时候使用“sudo”命令即可。

要切换回原来的用户，使用命令“sudo su 用户名”即可，比如我要从 root 切换回 zuozhongkai 这个用户，操作如图 2.2.4.9 所示：

```
root@ubuntu:/home/zuozhongkai# sudo su zuozhongkai
zuozhongkai@ubuntu:~$
```

图 2.2.4.9 切换回原来用户

10、显示文件内容命令 cat

查看文件内容是最常见的操作了，在 windows 下可以直接使用记事本查看一个文本文件内容，linux 下也有类似记事本的软件，叫做 gedit，找到一个文本文件，双击打开，默认使用的就是 gedit，如图 2.2.4.10 所示：

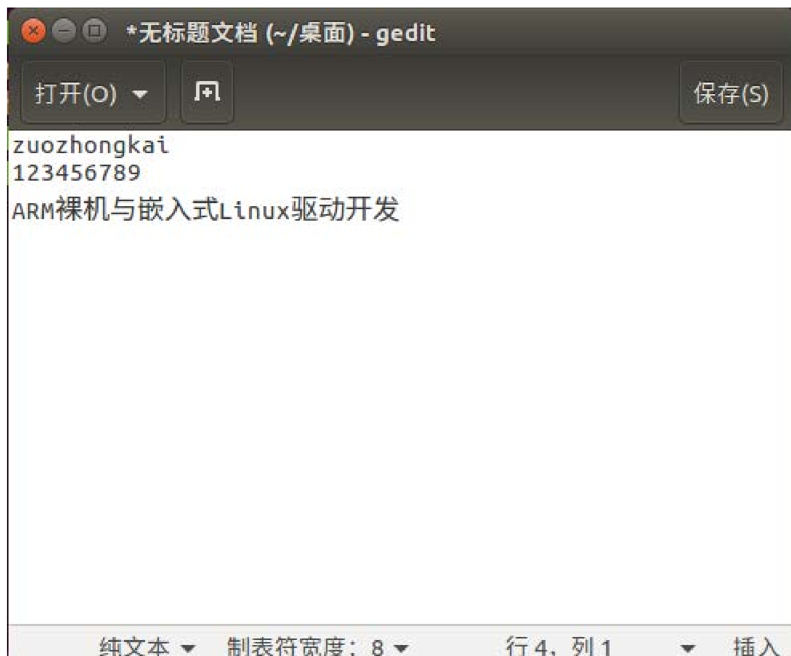


图 2.2.4.10 gedit 打开文档

我们现在讲解的是 Shell 命令，那么 Shell 下有没有办法读取文件的内容呢？肯定有的，那就是命令“cat”，命令格式如下：

```
cat  [选项]  [文件]
```

选项主要参数如下：

-n 由 1 开始对所有输出的行进行编号。

-b 和-n 类似，但是不对空白行编号。

-s 当遇到连续两个行以上空白行的话就合并为一个行空白行。

比如我们以查看文件“/etc/environment”的内容为例，结果如图 2.2.4.11 所示：

```
zuozhongkai@ubuntu:~$ cat /etc/environment
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games"
zuozhongkai@ubuntu:~$ cat /etc/environment -n
     1  PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games"
```

图 2.2.4.11 命令 cat 演示

11、显示和配置网络属性命令 ifconfig

ifconfig 是一个跟网络属性配置和显示密切相关的命令，通过此命令我们可以查看当前网络属性，也可以通过此命令配置网络属性，比如设置网络 IP 地址等等，此命令格式如下：

```
ifconfig  interface  options | address
```

主要参数如下：

interface 网络接口名称，比如 eth0 等。

up 开启网络设备。

down 关闭网络设备。

add IP 地址，设置网络 IP 地址。

netmask add 子网掩码。

命令 ifconfig 的使用如图 2.2.4.12 所示：

```
zuozhongkai@ubuntu:~$ ifconfig
ens33     Link encap:以太网  硬件地址 00:0c:29:96:89:d6
          inet 地址:192.168.31.235  广播:192.168.31.255  掩码:255.255.255.0
          inet6 地址: fe80::e92a:d882:e1b:7402/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:13404 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:967 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:810508 (810.5 KB)  发送字节:75728 (75.7 KB)

lo        Link encap:本地环回
          inet 地址:127.0.0.1  掩码:255.0.0.0
          inet6 地址: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  跃点数:1
          接收数据包:248 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:248 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:19004 (19.0 KB)  发送字节:19004 (19.0 KB)

zuozhongkai@ubuntu:~$ ifconfig ens33
ens33     Link encap:以太网  硬件地址 00:0c:29:96:89:d6
          inet 地址:192.168.31.235  广播:192.168.31.255  掩码:255.255.255.0
          inet6 地址: fe80::e92a:d882:e1b:7402/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:13406 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:971 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:810658 (810.6 KB)  发送字节:76218 (76.2 KB)
```

图 2.2.4.12 ifconfig 命令演示

在图 2.2.4.12 中有两个网卡：ens33 和 lo，ens33 是我的电脑实际使用的网卡，lo 是回测网卡。可以看出网卡 ens33 的 IP 地址为 192.168.31.235，我们使用命令“ifconfig”将网卡 ens33 的 IP 地址改为 192.168.31.20，操作如图 2.2.4.13 所示：

```
zuozhongkai@ubuntu:~$ sudo ifconfig ens33 192.168.31.20
zuozhongkai@ubuntu:~$ ifconfig ens33
ens33     Link encap:以太网  硬件地址 00:0c:29:96:89:d6
          inet 地址:192.168.31.20  广播:192.168.31.255  掩码:255.255.255.0
          inet6 地址: fe80::e92a:d882:e1b:7402/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:15188 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:1040 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:917930 (917.9 KB)  发送字节:84590 (84.5 KB)
```

图 2.2.4.13 修改网卡 IP 地址

从图 2.2.4.13 可以看出，我在使用命令“ifconfig”修改网卡 ens33 的 IP 地址的时候使用了“sudo”，说明在 Ubuntu 下修改网卡 IP 地址是需要 root 用户权限的。当修改完以后使用命令“ifconfig ens33”再次查看网卡 ens33 的命令，发现网卡 ens33 的 IP 地址变成了 192.168.31.20

12、系统帮助命令 man

Ubuntu 系统中有很多命令，这些命令都有不同的格式，不同的格式对应不同的功能，要完全记住这些命令和格式几乎是不可能的，必须有一个帮助手册，当我们需要了解一个命令的详细信息的时候查阅这个帮助手册就行了。Ubuntu 提供了一个命令来帮助用户完成这个功能，那就是“man”命令，通过“man”命令可以查看其它命令的语法格式、主要功能、主要参数说明

等，“man”命令格式如下：

```
man [命令名]
```

比如我们要查看命令“ifconfig”的说明，输入“man ifconfig”即可，如图 2.2.4.14 所示：

```
zuozhongkai@ubuntu:~$ man ifconfig
```

图 2.2.4.14 man 命令演示

在终端中输入图 2.2.4.14 所示的命令，然后点击回车键就会打开“ifconfig”这个命令的详细说明，如图 2.2.4.15 所示：

```
IFCONFIG(8)                 Linux Programmer's Manual                IFCONFIG(8)

NAME
       ifconfig - configure a network interface

SYNOPSIS
       ifconfig [-v] [-a] [-s] [interface]
       ifconfig [-v] interface [aftype] options | address ...

DESCRIPTION
       Ifconfig  is  used to configure the kernel-resident network interfaces.
       It is used at boot time to set up interfaces as necessary.  After that,
       it  is  usually  only  needed  when  debugging or when system tuning is
       needed.

       If no arguments are given, ifconfig displays the  status  of  the  cur-
       rently  active interfaces.  If a single interface argument is given, it
       displays the status of the given interface only; if a single  -a  argu-
       ment  is  given,  it  displays the status of all interfaces, even those
       that are down.  Otherwise, it configures an interface.

Address Families
       If the first argument after the interface name  is  recognized  as  the
 Manual page ifconfig(8) line 1 (press h for help or q to quit)
```

图 2.2.4.15 命令“ifconfig”详细介绍信息

图 2.2.4.15 就是命令“ifconfig”的详细介绍信息，按“q”键退出到终端。

13、 系统重启命令 reboot

通过点击 Ubuntu 主界面右上角的齿轮按钮来选择关机或者重启系统，同样的我们也可以使用 Shell 命令“reboot”来重启系统，直接输入命令“reboot”然后点击回车键接口，如图 2.2.4.16 所示：

```
zuozhongkai@ubuntu:~$ reboot
```

图 2.2.4.16 reboot 命令演示

14、系统关闭命令 poweroff

使用命令“reboot”可以重启系统，使用命令“poweroff”就可以关闭系统，在终端中输入命令“poweroff”然后按下回车键即可关闭 Ubuntu 系统，如图 2.2.4.17 所示：

```
zuozhongkai@ubuntu:~$ poweroff
```

图 2.2.4.17 poweroff 命令演示

15、软件安装命令 install

截至目前，我们都没有讲过 Ubuntu 下如何安装软件，因为 Ubuntu 安装软件不像 Windows 下那样，直接双击.exe 文件就开始安装了。Ubuntu 下很多软件是需要先自行下载源码，下载源

码以后自行编译，编译完成以后使用命令“intsall”来安装。当然 Ubuntu 下也有其它的软件安装方法，但是用的最多的就是自行编译源码然后安装，尤其是嵌入式 Linux 开发。命令“install”格式如下：

```
    install [选项]...   [-T]   源文件      目标文件
或：install [选项]...          源文件...   目录
或：install [选项]...    -t    目录        源文件...
或：install [选项]...    -d    目录...
```

“install”命令是将文件(通常是编译后的文件)复制到目的位置，在前三种形式中，将源文件复制到目标文件或将多个源文件复制到一个已存在的目录中同时设置其所有权和权限模式。在第四种形式会创建指定的目录。命令“install”通常和命令“apt-get”组合在一起使用的，关于“apt-get”命令我们稍后会讲解。

以上就是 Shell 最基本一些命令，还有一些其它的命令我们在后面在讲解，循序渐进嘛。

2.4 APT 下载工具

对于长时间使用 Windows 的我们，下载安装软件非常容易，Windows 下有很多的下载软件，Ubuntu 同样有不少的下载软件，本节我们讲解 Ubuntu 下我们用的最多的下载工具：APT 下载工具，APT 下载工具可以实现软件自动下载、配置、安装二进制或者源码的功能。APT 下载工具和我们前面讲解的“install”命令结合在一起构成了 Ubuntu 下最常用的下载和安装软件方法。它解决了 Linux 平台下一安装软件的一个缺陷，即软件之间相互依赖。

APT 采用的 C/S 模式，也就是客户端/服务器模式，我们的 PC 机作为客户端，当需要下载软件的时候就向服务器请求，因此我们需要知道服务器的地址，也叫做安装源或者更新源。打开系统设置，打开“软件和更新”设置，打开以后如图 2.4.1.1 所示：



图 2.4.1.1 软件和更新设置

在图 2.4.1.1 中的“Ubuntu 软件”选项卡下面的“下载自”就是 APT 工具的安装源，因为

我们是在中国，所以需要选择中国的服务器，否则的话可能会导致下载失败！这个也就是网上说的 Ubuntu 安装成功以后要更新源。

在我们使用 APT 工具下载安装或者更新软件的时候，首先会在下载列表中与本机软件对比，看一下需要下载哪些软件，或者升级哪些软件，默认情况下 APT 会下载最新的软件包，被安装的软件包所依赖的其它软件也会被下载安装。说了这么多，APT 下载工具究竟怎么用呢？APT 工具常用的命令如下：

1、更新本地数据库

如果想查看本地哪些软件可以更新的话可以使用如下命令：

```
sudo apt-get update
```

这个命令会访问源地址，并且获取软件列表并保存在本电脑上，过程如图 2.4.1.2 所示：

```
zuozhongkai@ubuntu:~$ sudo apt-get update
[sudo] zuozhongkai 的密码：
命中:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
命中:2 http://cn.archive.ubuntu.com/ubuntu xenial InRelease
获取:3 http://cn.archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
获取:4 http://cn.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
已下载 216 kB，耗时 13秒 (16.5 kB/s)
正在读取软件包列表... 完成
```

图 2.4.1.2 更新本地数据库

2、检查依赖关系

有时候本地某些软件可能存在依赖关系，所谓依赖关系就是 A 软件依赖于 B 软件。通过如下命令可以查看依赖关系，如果存在依赖关系的话 APT 会提出解决方案：

```
sudo apt-get check
```

上述命令的执行结果如图 2.4.1.3 所示：

```
zuozhongkai@ubuntu:~$ sudo apt-get check
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
```

图 2.4.1.3 检查依赖关系

3、软件安装

这个是重点了，安装软件，使用如下命令：

```
sudo apt-get install package-name
```

可以看出上述命令是由“apt-get”和“install”组合在一起的，“package-name”就是要安装的软件名字，“apt-get”负责下载软件，“install”负责安装软件。比如我们要安装软件 Ubuntu 下的串口工具“minicom”，我们就可以使用如下命令：

```
sudo apt-get install minicom
```

执行上述命令以后就会自动下载和安装 minicom 软件，如图 2.4.1.3 所示：

```
zuozhongkai@ubuntu:~$ sudo apt-get install minicom
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
将会同时安装下列软件:
  lrzsz
下列【新】软件包将被安装:
  lrzsz minicom
升级了 0 个软件包，新安装了 2 个软件包，要卸载 0 个软件包，有 92 个软件包未被升
级。
需要下载 306 kB 的归档。
解压缩后会消耗 1,193 kB 的额外空间。
您希望继续执行吗？ [Y/n] y
获取:1 http://cn.archive.ubuntu.com/ubuntu xenial/universe amd64 lrzsz amd64 0.1
2.21-8 [73.8 kB]
获取:2 http://cn.archive.ubuntu.com/ubuntu xenial-updates/universe amd64 minicom
 amd64 2.7-1+deb8u1build0.16.04.1 [232 kB]
已下载 306 kB，耗时 3秒 (80.8 kB/s)
正在选中未选择的软件包 lrzsz。
(正在读取数据库 ... 系统当前共安装有 217399 个文件和目录。)
正准备解包 .../lrzsz_0.12.21-8_amd64.deb  ...
正在解包 lrzsz (0.12.21-8) ...
正在选中未选择的软件包 minicom。
正准备解包 .../minicom_2.7-1+deb8u1build0.16.04.1_amd64.deb  ...
正在解包 minicom (2.7-1+deb8u1build0.16.04.1) ...
正在处理用于 man-db (2.7.5-1) 的触发器 ...
正在设置 lrzsz (0.12.21-8) ...
正在设置 minicom (2.7-1+deb8u1build0.16.04.1) ...
```

图 2.4.1.3 安装 minicom 软件

图 2.4.1.3 就是安装 minicom 这个软件的过程，在图 2.4.1.3 中安装的过程中，会有如下所示询问：

```
您希望继续执行吗？[Y/n]
```

如果希望继续执行的话就输入 y，如果不希望继续执行的话就输入 n。安装完成以后我们直接在终端输入如下命令打开 minicom 这个串口软件：

```
minicom -s
```

打开以后的 minicom 软件如图 2.4.1.4 所示：

```
+-----[configuration]------+
| Filenames and paths      |
| File transfer protocols  |
| Serial port setup        |
| Modem and dialing        |
| Screen and keyboard      |
| Save setup as dfl        |
| Save setup as..          |
| Exit                     |
| Exit from Minicom        |
```

图 2.4.1.4 minicom 软件

关于 minicom 的使用大家可以上网搜索一下，这里就不详细讲解了，要退出 minicom 可以直接按下 ESC 键

4、软件更新

有时候我们需要更新软件，更新软件的话使用命令：

```
sudo apt-get upgrade package-name
```

其中 package-name 为要升级的软件名字，比如我们升级刚刚安装的 minicom 这个软件，如图 2.4.1.5 所示：

```
zuozhongkai@ubuntu:~$ sudo apt-get upgrade minicom
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
minicom 已经是最新版 (2.7-1+deb8u1build0.16.04.1)。
正在计算更新... 完成
下列软件包的版本将保持不变：
  ubuntu-minimal
```

图 2.4.1.5 更新 minicom 软件

从图 2.4.1.5 可以看出，minicom 已经是最新的了，不用更新，不过有其它软件需要更新，因此会自动更新其它的软件。

5、卸载软件

如果要卸载某个软件的话使用如下命令：

```
sudo apt-get remove package-name
```

其中 package-name 是要卸载的软件，比如卸载前面安装的 minicom 这个软件，操作如图 2.4.1.6 所示：

```
zuozhongkai@ubuntu:~$ sudo apt-get remove minicom
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  lrzsz
使用'sudo apt autoremove'来卸载它(它们)。
下列软件包将被【卸载】：
  minicom
升级了 0 个软件包，新安装了 0 个软件包，要卸载 1 个软件包，有 1 个软件包未被升级。
解压缩后将会空出 928 kB 的空间。
您希望继续执行吗？ [Y/n] y
(正在读取数据库 ... 系统当前共安装有 217494 个文件和目录。)
正在卸载 minicom (2.7-1+deb8u1build0.16.04.1) ...
正在处理用于 man-db (2.7.5-1) 的触发器 ...
```

图 2.4.1.6 卸载软件

从图 2.4.1.6 中可以看出软件 minicom 被卸载掉了。关于 APT 下载工具就讲解到这里，我们用的最多的就是“sudo apt-get install package-name”来下载和安装软件。有关 Ubuntu 其它的安装软件的方法打开可以自行上网查阅学习，这里就不一一详解了。

2.5 Ubuntu 下文本编辑

2.5.1 Gedit 编辑器

进行文本编辑是最常用的操作，Windows 下我们会使用记事本来完成，或者其它一些优秀的文本编辑器，比如 notepad++，Ubuut 下有一个自带的文本编辑器，那就是 Gedit。Gedit 是一个窗口式的编辑器，关于 Gedit 的使用前面我们已经讲解了。本节我们重点讲解的是另外一个

编辑器：VI/VIM 编辑器。

2.5.2 VI/VIM 编辑器

我们如果要在终端模式下进行文本编辑或者修改文件就可以使用 VI/VIM 编辑器，Ubuntu 自带了 VI 编辑器，但是 VI 编辑器对于习惯了 Windows 下进行开发的人来说不方便，比如竟然不能使用键盘上的上下左右键调整光标位置。因此我推荐大家使用 VIM 编辑器，VIM 编辑器是 VI 编辑器升级版本，VI/VIM 编辑器都是一种基于指令式的编辑器，不需要鼠标，也没有菜单，仅仅使用键盘来完成所有的编辑工作。

我们需要先安装 VIM 编辑器，命令如下：

```
sudo apt-get install vim
```

安装完成以后就可以使用 VIM 编辑器了，VIM 编辑器有 3 种工作模式：输入模式、指令模式和底行模式，通过切换不同的模式可以完成不同的功能，我们就以编辑一个文本文档为例讲解 VIM 编辑器的使用。打开终端，输入命令：vi test.txt，如图 2.5.2.1 所示：

```
zuozhongkai@ubuntu:~$ vim test.txt
```

图 2.5.2.1 新建 test.txt 文档

在终端中输入图 2.5.2.1 中所示的命令以后就会创建一个 test.txt 文档，并且用 VIM 打开了，如图 2.5.2.2 所示：

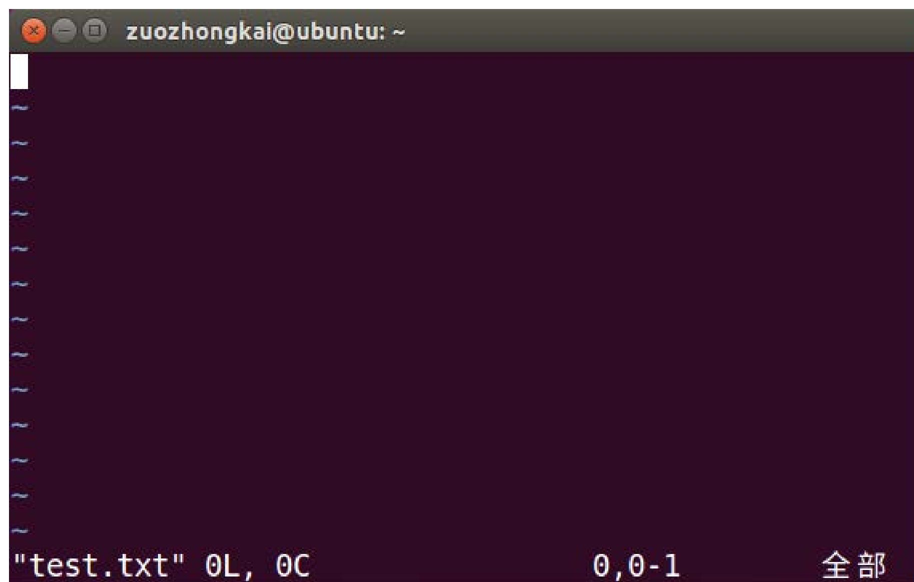


图 2.5.2.2 VIM 打开的 test.txt 文档

我们试着在图 2.5.2.2 中输入数字，发现根本没法输入，这不是因为你的键盘坏了。因为 VIM 默认是以只读模式打开的文档，因此我们要切换到输入模式，切换到输入模式的命令如下：

i　在当前光标所在字符的前面，转为输入模式。
I　在当前光标所在行的行首转换为输入模式。
a　在当前光标所在字符的后面，转为输入模式。
A　在光标所在行的行尾，转换为输入模式。
o　在当前光标所在行的下方，新建一行，并转为输入模式。
O　在当前光标所在行的上方，新建一行，并转为输入模式。
s　删除光标所在字符。
r　替换光标处字符。

最常用的就是“a”，我们在图 2.5.2.2 中按下键盘上的“a”键，这时候终端左下角会提示“插入”字样，表示我们进入到了输入模式，如图 2.5.2.3 所示：

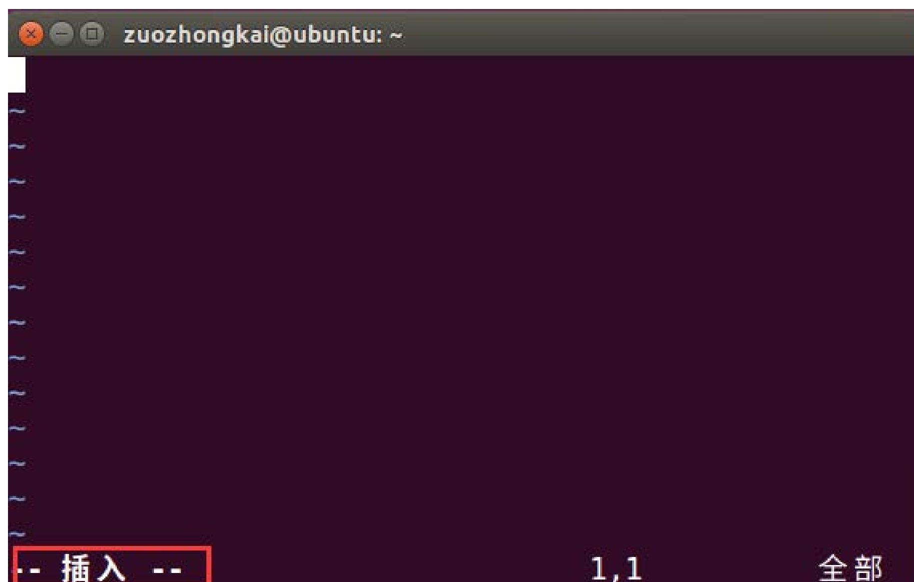


图 2.5.2.3 切换到插入模式

图 2.5.2.3 表明我们可以正常输入文本了，我们可以输入图 2.5.2.4 所示文本：

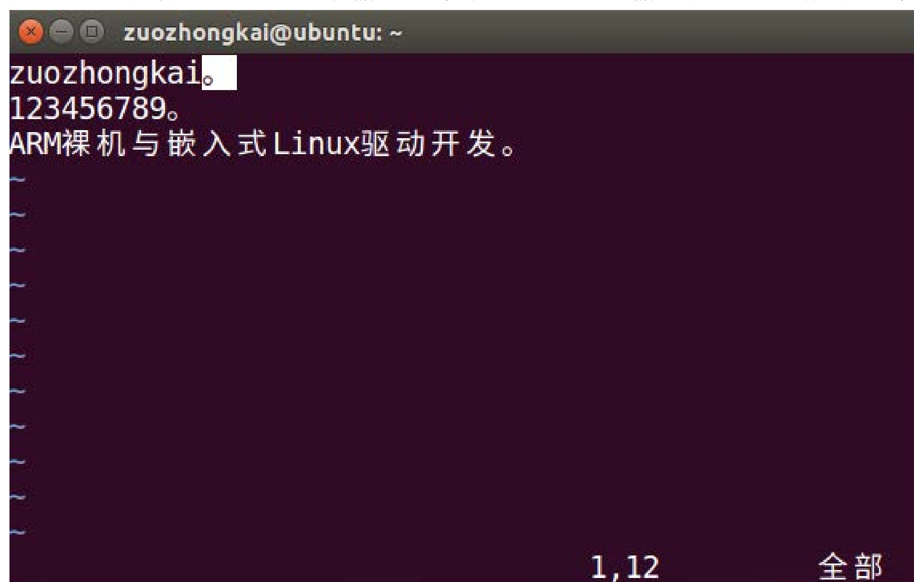


图 2.5.2.4 输入文本

在图 2.5.2.4 中我们在 test.txt 中输入了字母、数字和中文，我们输入完成以后需要保存文本啊，Windows 下的记事本可以使用快捷键 Ctrl+S 来保存，VIM 是否也可以使用 Ctrl+S 来保存呢？你会发现当你按下 Ctrl+S 键以后你的终端不能操作了！！！这是因为在 Ubuntu 下 Ctrl+S 快捷键不是用来完成保存的功能的，而是暂停该终端！所以你一旦在使用终端的时候按下 Ctrl+S 快捷键，那么你的终端肯定不会再有任何反应，如果你按下 Ctrl+S 关闭了当前终端的话可以按下 Ctrl+Q 来重新打开终端。

既然 Ctrl+S 不能保存文本文档，那么有没有其它方法保存文本文档呢？肯定是有的，我们需要从 VIM 现在的输入模式切换到指令模式，方式就是按下键盘的 ESC 键，按下 ESC 键以后终端坐下角的“插入”字样就会消失，此时你就不能在输入任何文本了，如果想再次输入文本的话就按下“a”键重新进入到输入模式。指令模式顾名思义就是输入指令的模式，这些指令是

控制文本的指令，我们将这些指令进行分类，如下所示：

1、移动光标指令：

h(或左方向键) 光标左移一个字符。
l(或右方向键) 光标右移一个字符。
j(或下方向键) 光标下移一行。
k(或上方向键) 光标上移一行。
nG 光标移动到第 n 行首。
n+ 光标下移 n 行。
n- 光标上移 n 行。

2、屏幕翻滚指令

Ctrl+f 屏幕向下翻一页，相当于下一页。
Ctrl+b 屏幕向上翻一页，相当于上一页。

3、复制、删除和粘贴指令

cc 删除整行，并且修改整行内容。
dd 删除该行，不提供修改功能。
ndd 删除当前行向下 n 行。
x 删除光标所在的字符。
X 删除光标前面的一个字符。
nyy 复制当前行及其下面 n 行。
p 粘贴最近复制的内容。

上面就是 VI/VIM 的命令模式下最常用的一些命令，还有一些不常用的我没有列出来，感兴趣的可以自行上网查阅。从上面的命令可以看出，并没有保存文本的命令，那是因为保存文档的命令是在底行模式中，我们要先进入到指令模式，进入底行模式的方式是先进入指令模式下，然后在指令模式下输入“：”进入底行模式，如图 2.5.2.5 所示：

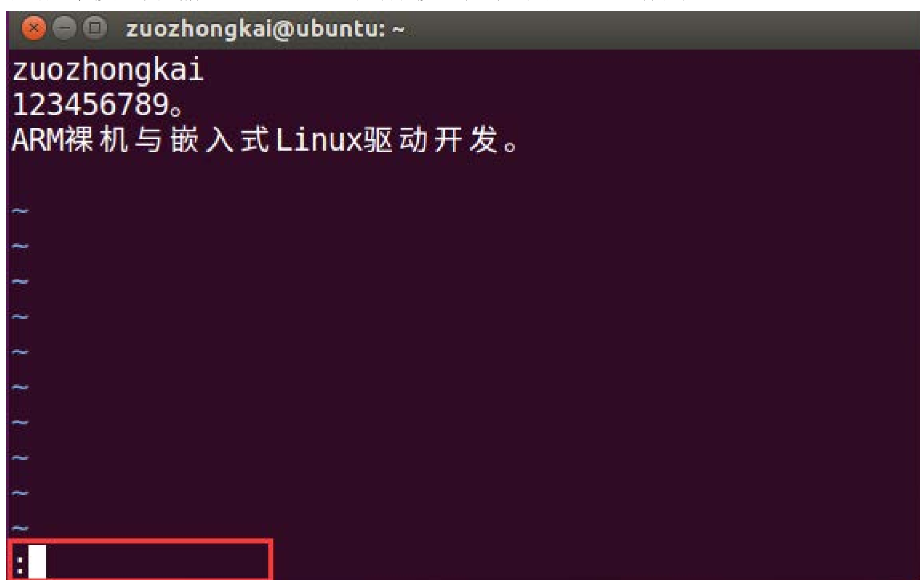


图 2.5.2.5 “:”底行模式

在图 2.5.2.5 中当进入底行模式以后会在终端的左下角就会出现符号“：”，我们可以在“：”后面输入命令，常用的命令如下：

x 保存当前文档并且退出。

q 退出。

w 保存文档。

q! 退出 VI/VIM，不保存文档。

如果我们要退出并保存文本的话需要在“：”底行模式下输入“wq”，如图 2.5.2.6 所示：

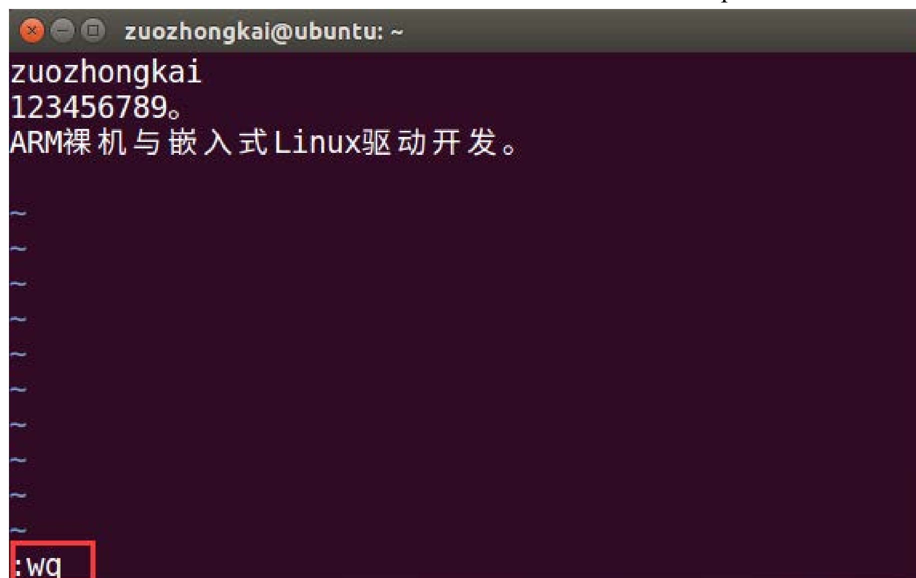


图 2.5.2.6 保存并退出 VIM

在“：”底行模式下输入“wq”以后按下回车键就保存 test.txt 并退出 VI/VIM 编辑器，退出以后我们可以使用命令“cat”来查看刚刚新建的 test.txt 文档的内容，如图 2.5.2.7 所示：

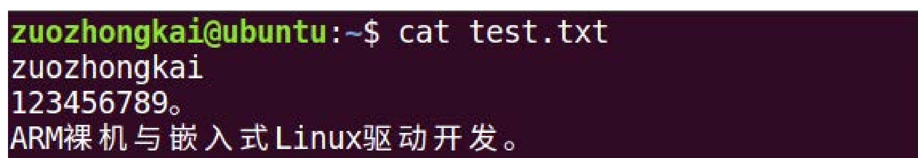


图 2.5.2.7 查看文档内容

从图 2.5.2.7 中可以看出，test.txt 中的内容就是我们用 VIM 输入的内容，至此我们就完整的进行了一遍 VI/VIM 创建文档、编辑文档和保存文档。

在上面讲解进入 VIM 的底行模式的时候之说了在指令模式下输入“：”的方法，还可以在指令模式下输入“/”进入底行模式，输入“/”以后如图 2.5.2.8 所示。

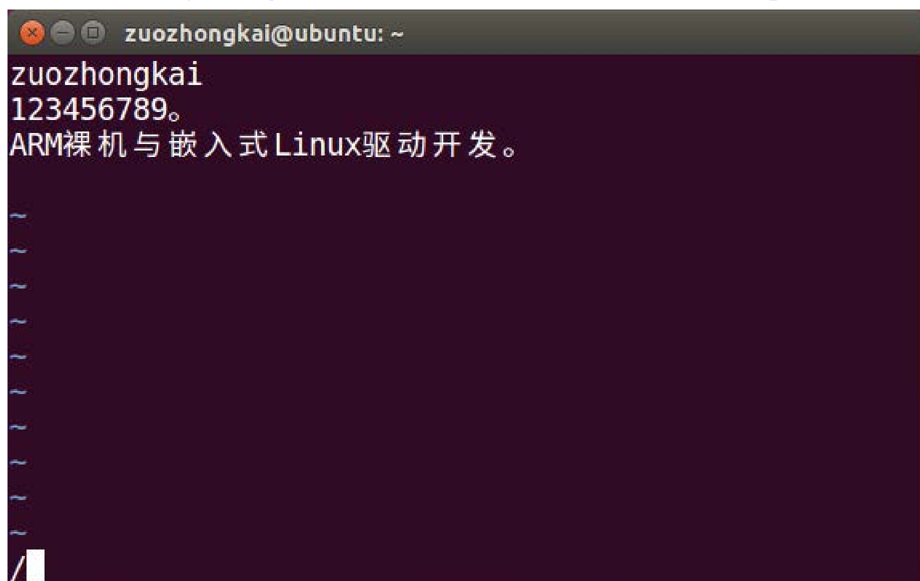


图 2.5.2.8 “/”底行模式

在“/”底行模式下我们可以在文本中搜索指定的内容，比如搜索 test.txt 文件中“嵌入式”三个字，使用方法如图 2.5.2.9 所示：

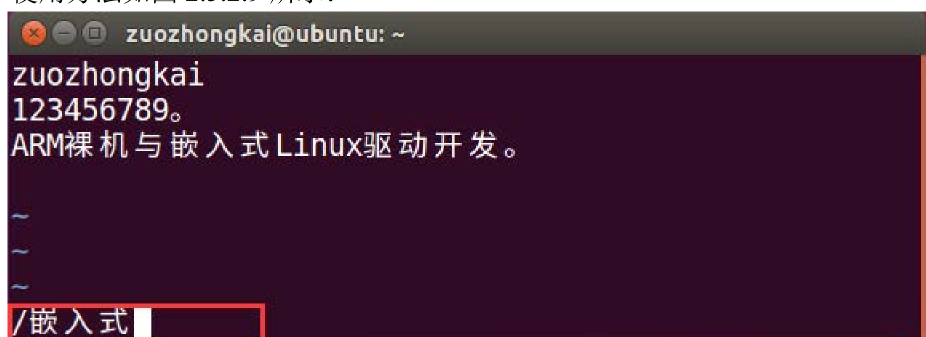


图 2.5.2.9 搜索文本

在“/”后面输入要搜索的内容，然后按下回车键就会在 test.txt 中找到与字符串“嵌入式”匹配的部分，如图 2.5.2.10 所示：

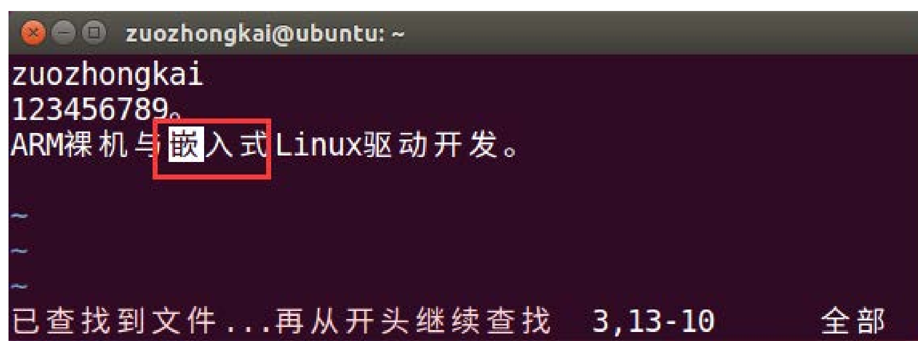


图 2.5.2.10 查找到指定内容

图 2.5.2.10 中可以看出，在 test.txt 中找到了“嵌入式”这个词，并且标记出来位置。我们以后要在一个文档中搜索是否存在某个字符串的时候就可以使用这种方法。有关 VI/VIM 编辑器的讲解就到这里，我们完整的练习了一遍如何使用 VIM 创建文档、编辑文档、保存文档和在文档中搜索字符串。有关更多更详细的 VIM 编辑器的操作大家自行上网查阅相关文档和博客。

2.6 Linux 文件系统

操作系统的基本功能之一就是文件管理，而文件的管理是由文件系统来完成的。Linux 支持多种文件系统，本节我们就来讲解 Linux 下的文件系统、文件系统类型、文件系统结构和文件系统相关 Shell 命令。

2.6.1 Linux 文件系统简介以及类型

1、Linux 文件系统简介

操作系统就是处理各种数据的，这些数据在硬盘上就是二进制，人类肯定不能直接看懂这些二进制数据，要有一个翻译器，将这些二进制的数据还原为人类能看懂的文件形式，这个工作就是由文件系统来完成的，文件系统的目的就是实现数据的查询和存储，由于使用场合、使用环境的不同，Linux 有多种文件系统，不同的文件系统支持不同的体系。文件系统是管理数据的，而可以存储数据的物理设备有硬盘、U 盘、SD 卡、NAND FLASH、NOR FLASH、网络存储设备等。不同的存储设备其物理结构不同，不同的物理结构就需要不同的文件系统去管理，比如管理 NAND FLASH 的话使用 YAFFS 文件系统，管理硬盘、SD 卡的话就是 ext 文件系统等等。

我们在使用 Windows 的时候新买一个硬盘回来一般肯定是将这个硬盘分为好几个盘，比如 C 盘、D 盘等等。这个叫磁盘的分割，Linux 下也支持磁盘分割，Linux 下常用的磁盘分割工具为：fdisk，fdisk 这个工具我们后面会详细讲解怎么用，因为我们移植 Linux 的时候需要将 SD 卡分为三个分区来存储不同的东西。在 Windows 下我们创建一个新的盘符以后都要做格式化处理，格式化其实就是给这个盘符创建文件系统的过程，我们在 Windows 格式化某个盘的时候都会让你选择文件系统，如图 2.6.1.1 所示：

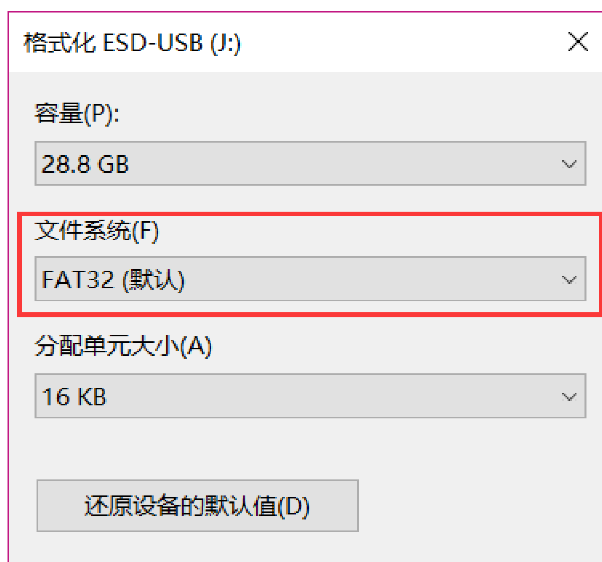


图 2.6.1.1 格式化磁盘

图 2.6.1.1 就是格式化磁盘的时候选择文件系统，Windows 下一般有 FAT、NTFS 和 exFAT 这些文件系统。同样的，在 Linux 下我们使用 fdisk 创建好分区以后也是要先在创建好的分区上面创建文件系统，也就是格式化。

在 Windows 下有磁盘分区的概念，比如 C，D，E 盘等，在 Linux 下没有这个概念，因此 Linux 下你找不到像 C、D、E 盘这样的东西。前面我们说了 Linux 下可以给磁盘分割，但是没

有 C、D、E 盘那怎么访问这些分区呢？在 Linux 下创建一个分区并且格式化好以后我们要将其“挂载”到一个目录下才能访问这个分区。Windows 的文件系统挂载过程是其内部完成的，用户是看不到的，Linux 下我们使用 mount 命令来挂载磁盘。挂载磁盘的时候是需要确定挂载点的，也就是你的这个磁盘要挂载到哪个目录下。

2、Linux 文件系统类型

前面我们说了，在 Windows 下有 FAT、NTFS 和 exFAT 这样的文件系统，在 Linux 下又有哪些文件系统呢，Linux 下的文件系统主要有 ext2、ext3、ext4 等文件系统。Linux 还支持其他的 UNIX 文件系统，比如 XFS、JFS、UFS 等，也支持 Windows 的 FAT 文件系统和网络文件系统 NFS 等。这里我们主要讲一下 Linux 自带的 ext2、ext3 和 ext4 文件系统。

ext2 文件系统：

ext2 是 Linux 早期的文件系统，但是随着技术的发展 ext2 文件系统已经不推荐使用了，ext2 是一个非日志文件系统,大多数的 Linux 发行版都不支持 ext2 文件系统了。

ext3 文件系统：

ext3 是在 ext2 的基础上发展起来的文件系统，完全兼容 ext2 文件系统，ext3 是一个日志文件系统，ext3 支持大文件，ext3 文件系统的特点有如下：

高可靠性：使用 ext3 文件系统的话，即使系统非正常关机、发生死机等情况，恢复 ext3 文件系统也只需要数十秒。

数据完整性：ext3 提高了文件系统的完整性，避免意外死机或者关机对文件系统的伤害。

文件系统速度：ext3 的日志功能对磁盘驱动器读写头进行了优化，文件系统速度相对与 ext2 来说没有降低。

数据转换：从 ext2 转换到 ext3 非常容易，只需要两条指令就可以完成转换。用户不需要花时间去备份、恢复、格式化分区等，用 ext3 文件系统提供的工具 tune2fs 即可轻松的将 ext2 文件系统转换为 ext3 日志文件系统。ext3 文件系统不需要经过任何修改，可以直接挂载成 ext2 文件系统。

ext4 文件系统：

ext4 文件系统是在 ext3 上发展起来的，ext4 相比与 ext3 提供了更佳的性能和可靠性，并且功能更丰富，ext4 向下兼容 ext3 和 ext2，因此可以将 ext2 和 ext3 挂载为 ext4。那么我们安装的 Ubuntu 使用的哪个版本的文件系统呢？在终端中输入如下命令来查询当前磁盘挂载的啥文件系统：

```
df  -T  - h
```

结果如图 2.6.1.2 所示：

```
zuozhongkai@ubuntu:/$ df -T -h
文件系统        类型      容量  已用  可用 已用% 挂载点
udev           devtmpfs  3.9G     0  3.9G    0% /dev
tmpfs          tmpfs     796M  9.3M  787M    2% /run
/dev/sda1      ext4      192G  5.2G  177G    3% /
tmpfs          tmpfs     3.9G  252K  3.9G    1% /dev/shm
tmpfs          tmpfs     5.0M  4.0K  5.0M    1% /run/lock
tmpfs          tmpfs     3.9G     0  3.9G    0% /sys/fs/cgroup
tmpfs          tmpfs     796M   56K  796M    1% /run/user/1000
```

图 2.6.1.2 Ubuntu 使用的文件系统

在图 2.6.1.2 中，框起来的就是我们安装 Ubuntu 的这个磁盘，在 Linux 下一切皆为文件，“/dev/sda1”就是我们的磁盘分区，可以看出这个磁盘分区类型是 ext4，它的挂载点是“/”，也

就是根目录。

2.6.2 Linux 文件系统结构

在 Windows 下直接打开 C 盘，我们进入的就是 C 盘的根目录，打开 D 盘进入的就是 D 盘的根目录，比如 C 盘根目录如下：

› 本地磁盘 (C:)

名称	修改日期	类型	大小
$WINDOWS.~BT	2017-01-18 10:25	文件夹	
$Windows.~WS	2017-12-17 11:34	文件夹	
AppData	2018-08-27 14:47	文件夹	
DRMsoft	2017-09-07 11:24	文件夹	
ESD	2017-12-17 12:11	文件夹	
Intel	2016-09-07 15:13	文件夹	
M1530_MFP_Series_Basic_Solution	2016-12-29 23:30	文件夹	
PerfLogs	2016-07-16 19:47	文件夹	
Program Files	2017-12-13 15:08	文件夹	
Program Files (x86)	2018-12-18 0:04	文件夹	
ProgramData	2018-12-17 21:57	文件夹	
touchgfx-env	2017-12-05 11:37	文件夹	
TouchGFXProjects	2018-01-07 11:22	文件夹	
uacdump	2016-11-28 12:39	文件夹	
Users	2016-09-08 13:00	文件夹	
Windows	2018-10-31 23:49	文件夹	
InstallConfig.ini	2017-10-13 23:12	配置设置	1 KB

图 2.6.2.1 C 盘根目录

在 Linux 下因为没有 C、D 盘之说，因此 Linux 只有一个根目录，没有 C 盘根目录、D 盘根目录之类的。其实如果你的 Windows 只有一个 C 盘的话那么整个系统也就只有一个根目录。Windows 下的 C 盘根目录就是“C:”,在 Linux 下的根目录就是“/”，你没有看错，Linux 根目录就是用“/”来表示的，打开 Ubuntu 的文件浏览器，文件浏览器在左侧的导航栏，图标如图 2.6.2.2 所示：



图 2.6.2.2 文件浏览器

打开以后的文件浏览器如图 2.6.2.3 所示：

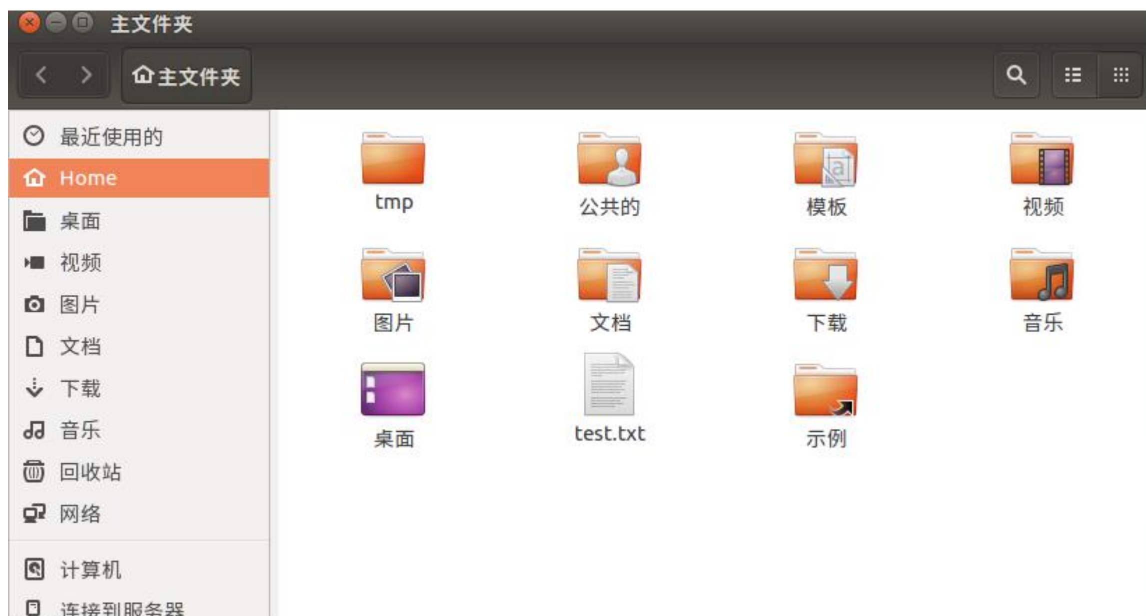


图 2.6.2.3 文件浏览器

直接打开文件浏览器以后，我们默认不是处于根目录中的，不像 Windows，我们直接打开 C 盘就处于 C 盘根目录下。Ubuntu 是支持多用户的，Ubuntu 为每个用户创建了一个根目录，比如我电脑现在登陆的是“zuozhongkai”这个用户，因此默认进入的是“zuozhongkai”这个用户的根目录。我们点击图 2.6.2.3 中左侧的“计算机”，打开以后如图 2.6.2.4 所示：



图 2.6.2.4 根目录“/”

图 2.6.2.4 就是 Ubuntu 的根目录“/”，这时候肯定就有人有疑问，刚刚说 Ubuntu 会给每个

用户创建一个根目录，那这些用户的根目录在哪里？是不是和根目录“/”是一个地位的？其实所谓的给每个用户创建一个根目录只是方便说而已，这个所谓的用户根目录其实就是“/”下的一个文件夹，以我的“zuozhongkai”这个用户为例，其用户根目录就是：/home/zuozhongkai。只要你创建了一个用户，那么系统就会在/home 这个目录下创建一个以这个用户名命名的文件夹，这个文件夹就是这个用户的根目录。

用户可以对自己的用户根目录下的文件进行随意的读写操作，但是如果要修改根目录“/”下的文件就会提示没有权限。打开终端以后默认进入的是当前用户根目录，比如我们打开终端以后输入“ls”命令查看当前目录下有什么文件，结果如图 2.6.2.5 所示：

```
zuozhongkai@ubuntu:~$ ls
examples.desktop  tmp     模板  图片  下载  桌面
test.txt          公共的  视频  文档  音乐
```

图 2.6.2.5 目录查看

可以看出图 2.6.2.5 中的文件和图 2.6.2.3 中的一模一样，都是“zuozhongkai”这个账户的根目录。我们来看一下根目录“/”下都有哪些文件，在终端中输入如下命令：

```
cd  /       //进入到根目录“/
ls          //查看根目录“/”下的文件以及文件夹
```

执行上述两行命令以后，终端如图 2.6.2.6 所示：

```
zuozhongkai@ubuntu:~$ cd /
zuozhongkai@ubuntu:/$ ls
bin    dev   initrd.img      lib64       mnt   root  snap  tmp  vmlinuz
boot   etc   initrd.img.old  lost+found  opt   run   srv   usr  vmlinuz.old
cdrom  home  lib             media       proc  sbin  sys   var
```

图 2.6.2.6 查看根目录“/”

图 2.6.2.6 中列举出了根目录“/”下面的所有文件夹，这里我们仔细观察一下，当我们进入到根目录“/”里面以后终端提示符“$”前面的符号“~”变成了“/”，这是因为当我们在终端中切换了目录以后“$”前面就会显示切换以后的目录路径。我们来看一下根目录“/”中的一些重要的文件夹：

/bin 存储一些二进制可执行命令文件，/usr/bin 也存放了一些基于用户的命令文件。
/sbin 存储了很多系统命令，/usr/sbin 也存储了许多系统命令。
/root 超级用户 root 的根目录文件。
/home 普通用户默认目录，在该目录下，每个用户都有一个以本用户名命名的文件夹。
/boot 存放 Ubuntu 系统内核和系统启动文件。
/mnt 通常包括系统引导后被挂载的文件系统的挂载点。
/dev 存放设备文件，我们后面学习 Linux 驱动主要是跟这个文件夹打交道的。
/etc 保存系统管理所需的配置文件和目录。
/lib 保存系统程序运行所需的库文件，/usr/lib 下存放了一些用于普通用户的库文件。
/lost+found 一般为空，当系统非正常关机以后，此文件夹会保存一些零散文件。
/var 存储一些不断变化的文件，比如日志文件
/usr 包括与系统用户直接有关的文件和目录，比如应用程序和所需的库文件。
/media 存放 Ubuntu 系统自动挂载的设备文件。
/proc 虚拟目录，不实际存储在磁盘上，通常用来保存系统信息和进程信息。
/tmp 存储系统和用户的临时文件，该文件夹对所有的用户都提供读写权限。
/opt 可选文件和程序的存放目录。
/sys 系统设备和文件层次结构，并向用户程序提供详细的内核数据信息。

2.6.2 文件操作命令

本节我们来学习一下在终端进行文件操作的一些常用命令：

1、创建新文件命令—touch

在前面学习 VIM 的时候我们知道可以用 vi 指令来创建一个文本文档，本节我们就学习一个功能更全面的文件创建命令—touch。touch 不仅仅可以用用来创建文本文档，其它类型的文档也可以创建，命令格式如下：

```
touch    [参数]   [文件名]
```

使用 touch 创建文件的时候，如果[文件名]的文件不存在，那就直接创建一个以[文件名]命名的文件，如果[文件名]文件存在的话就仅仅修改一下此文件的最后修改日期，常用的命令参数如下：

-a 只更改存取时间。

-c 不建立任何文件。

-d<日期> 使用指定的日期，而并非现在日期。

-t<时间> 使用指定的时间，而并非现在时间。

进入到用户根目录下，直接使用命令“cd ~”即可快速进入用户根目录，进入用户根目录以后使用 touch 命令创建一个名为 test 的文件，创建过程如图 2.6.2.1 所示：

```
zuozhongkai@ubuntu:~$ cd ~    //进入用户根目录
zuozhongkai@ubuntu:~$ ls      //查看当前目录下的文件
examples.desktop  tmp      模板  图片  下载  桌面
test.txt          公共的   视频  文档  音乐
zuozhongkai@ubuntu:~$ touch test  //创建test文件
zuozhongkai@ubuntu:~$ ls test     //查看创建的test文件
test
zuozhongkai@ubuntu:~$ ls test -l //查看创建的test文件详细信息
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月  22 01:24 test
```

图 2.6.2.1 touch 命令操作

2、文件夹创建命令—mkdir

既然可以创建文件，那么肯定也可以创建文件夹，创建文件夹使用命令“mkdir”，命令格式如下：

```
mkdir    [参数]   [文件夹名目录名]
```

主要参数如下：

-p 如所要创建的目录其上层目录目前还未创建，那么会一起创建上层目录。

我们在用户根目录下创建两个分别名为“testdir1”和“testdir2”的文件夹，操作如图 2.6.2.2 所示：

```
zuozhongkai@ubuntu:~$ ls  //查看当前目录下的文件
examples.desktop  test.txt  公共的  视频  文档  音乐
test              tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ mkdir testdir1 //创建testdir1文件夹
zuozhongkai@ubuntu:~$ mkdir testdir2 //创建testdir2文件夹
zuozhongkai@ubuntu:~$ ls            //查看当前目录下的所有文件，看看文件夹创建是否成功
examples.desktop  testdir1  test.txt  公共的  视频  文档  音乐
test              testdir2  tmp       模板    图片  下载  桌面
```

图 2.6.2.2 创建文件夹

在图 2.6.2.2 中，我们使用命令“mkdir”创建了“testdir1”和“testdir2”这两个文件夹。

3、文件及目录删除命令—rm

既然有创建文件的命令，那肯定有删除文件的命令，要删除一个文件或者文件夹可以使用命令“rm”，此命令可以完成删除一个文件或者多个文件及文件夹，它可以实现递归删除。对于链接文件，只删除链接，原文件保持不变，所谓的链接文件，其实就是 Windows 下的快捷方式文件，此命令格式如下：

```
rm  [参数]  [目的文件或文件夹目录名]
```

命令主要参数如下：

-d 直接把要删除的目录的硬连接数据删成 0，删除该目录。
-f 强制删除文件和文件夹(目录)。
-i 删除文件或者文件夹(目录)之前先询问用户。
-r 递归删除，指定文件夹(目录)下的所有文件和子文件夹全部删除掉。
-v 显示删除过程。

我们使用 rm 命令来删除前面使用命令“touch”创建的 test 文件，操作过程如图 2.6.2.3 所示：

```
zuozhongkai@ubuntu:~$ ls //查看当前目录下的所有文件
examples.desktop  testdir1  test.txt  公共的  视频  文档  音乐
test              testdir2  tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ rm test //删除文件test
zuozhongkai@ubuntu:~$ ls      //查看当前目录，看看test文件是否删除
examples.desktop  testdir2  tmp     模板  图片  下载  桌面
testdir1          test.txt  公共的  视频  文档  音乐
```

图 2.6.2.3 删除文件

命令“rm”也可以直接删除文件夹，我们可以试一下删除前面创建的 testdir1 文件夹，先直接使用命令“rm testdir1”测试一下是否可以删除，结果如图 2.6.2.4 所示：

```
zuozhongkai@ubuntu:~$ rm testdir1
rm: 无法删除'testdir1': 是一个目录
```

图 2.6.2.4 删除文件夹

在图 2.6.2.4 中可以看出，直接使用命令“rm”是无法删除文件夹(目录)的，我们需要加上参数“-rf”，也就是强制递归删除文件夹(目录)，操作结果如图 2.6.2.5 所示：

```
zuozhongkai@ubuntu:~$ ls  //查看当前目录下的所有文件
examples.desktop  testdir2  tmp     模板  图片  下载  桌面
testdir1          test.txt  公共的  视频  文档  音乐
zuozhongkai@ubuntu:~$ rm testdir1 -rf //使用参数“-rf”强制递归删除文件夹
zuozhongkai@ubuntu:~$ ls              //查看删除文件夹以后的当前目录
examples.desktop  test.txt  公共的  视频  文档  音乐
testdir2          tmp       模板    图片  下载  桌面
```

图 2.6.2.5 带参数删除文件夹

从图 2.6.2.5 可以看出，当在命令“rm”中加入参数“-rf”以后就可以删除掉文件夹“testdir1”了。

4、文件夹(目录)删除命令—rmdir

上面我们讲解了如何使用命令“rm”删除文件夹，那就是要加上参数“-rf”，其实 Linux 提供了直接删除文件夹(目录)的命令—rmdir，它可以不加任何参数的删除掉指定的文件夹(目录)，命令格式如下：

rmdir [参数] [文件夹(目录)]

命令主要参数如下：

-p 删除指定的文件夹(目录)以后，若上层文件夹(目录)为空文件夹(目录)的话就将其一起删除。

我们使用命令“rmdir”删除掉前面创建的“testdir2”文件夹，操作过程如图 2.6.2.6 所示：

```
zuozhongkai@ubuntu:~$ ls    //查看当前目录下的所有文件
examples.desktop  test.txt  公共的  视频  文档  音乐
testdir2          tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ rmdir testdir2 //删除文件夹testdir2
zuozhongkai@ubuntu:~$ ls            //查看删除文件夹以后的目录文件
examples.desktop  tmp     模板  图片  下载  桌面
test.txt          公共的  视频  文档  音乐
```

图 2.6.2.6 命令 rmdir 删除文件夹

5、文件复制命令—cp

在 Windows 下我们可以通过在文件上点击鼠标右键来进行文件的复制和粘贴，在 Ubuntu 下我们也可以通过点击文件右键进行文件的复制和粘贴。但是本节我们来讲解如何在终端下使用命令来进行文件的复制，Linux 下的复制命令为“cp”，命令描述如下：

cp [参数] [源地址] [目的地址]

主要参数描述如下：

-a 此参数和同时指定“-dpR”参数相同

-d 在复制有符号连接的文件时，保留原始的连接。

-f 强行复制文件，不管要复制的文件是否已经存在于目标目录。

-I 覆盖现有文件之前询问用户。

-p 保留源文件或者目录的属性。

-r 或-R 递归处理，将指定目录下的文件及子目录一并处理

我们在用户根目录下，使用前面讲解的命令“mkdir”创建两个文件夹：test1 和 test2，过程如图 2.6.2.7 所示。

```
zuozhongkai@ubuntu:~$ ls   //查看当前目录下的文件
examples.desktop  tmp     模板  图片  下载  桌面
test.txt          公共的  视频  文档  音乐
zuozhongkai@ubuntu:~$ mkdir test1 test2 //创建test1和test2两个文件夹
zuozhongkai@ubuntu:~$ ls               //查看当前目录下的所有文件
examples.desktop  test2     tmp     模板  图片  下载  桌面
test1             test.txt  公共的  视频  文档  音乐
```

图 2.6.2.7 创建 test1 和 test2 两个文件夹

进入上面创建的 test1 文件夹，然后在 test1 文件夹里面创建一个 a.c 文件，操作过程如图 2.6.2.8 所示：

```
zuozhongkai@ubuntu:~$ cd test1          //进入test1文件夹
zuozhongkai@ubuntu:~/test1$ touch a.c //创建a.c文件
zuozhongkai@ubuntu:~/test1$ ls        //查看当前目录下所有文件
a.c
```

图 2.6.2.8 创建 a.c 文件

我们先将图 2.6.2.8 中的 a.c 这个文件做个备份，也就是复制到同文件夹 test1 里面，新的文件命名为 b.c。然后在将 test1 文件夹中的 a.c 和 b.c 这两个文件都复制到文件夹 test2 中，操作

如图 2.6.2.9 所示

```
zuozhongkai@ubuntu:~/test1$ cp a.c b.c        //拷贝a.c到本文件夹中，并重命名为b.c
zuozhongkai@ubuntu:~/test1$ ls                //查看当前目录下的所有文件
a.c  b.c
zuozhongkai@ubuntu:~/test1$ cp *.c ../test2   //拷贝a.c和b.c到文件夹test2中
zuozhongkai@ubuntu:~/test1$ cd ../test2       //进入上级目录中的test2文件夹
zuozhongkai@ubuntu:~/test2$ ls                //查看test2文件夹下的文件
a.c  b.c
```

图 2.6.2.9 拷贝文件

在图 2.6.2.9 中，我们添加了一些高级使用技巧，首先是拷贝 a.c 和 b.c 文件到 test2 文件夹中，我们使用了通配符"*"，"*.c"就表示 test1 下的所有以".c"结尾的文件，也就是 a.c 和 b.c。"../test2"中的"../"表示上级目录，因此"../test2"就是上级目录下的 test2 文件夹。

上面都是文件复制，我们接下来学习一下文件夹复制，我们将 test2 文件夹复制到同目录下，新拷贝的文件夹命名为 test3，操作如图 2.6.2.10 所示：

```
zuozhongkai@ubuntu:~$ cp -rf test2/ test3/  //复制test2文件夹为test3文件夹
zuozhongkai@ubuntu:~$ ls                    //查看当前目录下的所有文件
examples.desktop  test2  test.txt  公共的  视频  文档  音乐
test1             test3  tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ cd test3              //进入test3文件夹
zuozhongkai@ubuntu:~/test3$ ls              //查看test3中的所有文件
a.c  b.c
```

图 2. 6. 2. 10 拷贝文件夹

6、文件移动命令—mv

有时候我们需要将一个文件或者文件夹移动到另外一个地方去，或者给一个文件或者文件夹进行重命名，这个时候我们就可以使用命令"mv"了，此命令格式如下：

```
mv [参数]  [源地址]    [目的地址]
```

主要参数描述如下：

-b 如果要覆盖文件的话覆盖前先进行备份。

-f 若目标文件或目录与现在的文件重复，直接覆盖目的文件或目录。

-I 在覆盖之前询问用户。

使用上面讲解"cp"命令的时候创建了三个文件夹，在上面创建的 test1 文件夹里面创建一个 c.c 文件，然后将 c.c 这个文件重命名为 d.c。最后将 d.c 这个文件移动到 test2 文件夹里面，操作如图 2.6.2.11 所示：

```
zuozhongkai@ubuntu:~$ cd test1              //进入test1文件夹
zuozhongkai@ubuntu:~/test1$ touch c.c       //创建c.c文件
zuozhongkai@ubuntu:~/test1$ ls              //查看当前目录下的所有文件
a.c  b.c  c.c
zuozhongkai@ubuntu:~/test1$ mv c.c d.c      //移动c.c到当前目录下，并重命名为d.c
zuozhongkai@ubuntu:~/test1$ ls              //查看当前目录下的所有文件
a.c  b.c  d.c
```

图 2.6.2.11 移动文件操作

我们再将 test1 中的 d.c 文件移动到 test2 文件夹里面，操作如图 2.6.2.12 所示：

```
zuozhongkai@ubuntu:~/test2$ ls          //查看test2下所有文件
a.c  b.c
zuozhongkai@ubuntu:~/test2$ cd ../test1     //进入test1文件夹中
zuozhongkai@ubuntu:~/test1$ ls              //查看test1文件夹所有文件
a.c  b.c  d.c
zuozhongkai@ubuntu:~/test1$ mv d.c ../test2 //将d.c移动到test2中
zuozhongkai@ubuntu:~/test1$ ls ../test2     //查看test2下的所有文件
a.c  b.c  d.c
```

图 2.6.2.12 移动文件操作

2.6.3 文件压缩和解压缩

文件的压缩和解压缩是非常常见的操作，在 Windows 下我们有很多压缩和解压缩的工具，比如 zip、360 压缩等等。在 Ubuntu 下也有压缩工具，本节我们学习 Ubuntu 下图形化以及命令行这两种压缩和解压缩操作。

1、图形化压缩和解压缩

图形化压缩和解压缩和 Windows 下基本一样，在要压缩或者解压的文件上点击鼠标右键，然后选择要进行的操作，我们先讲解一下如何进行文件的压缩。首先找到要压缩的文件，然后在要压缩的文件上点击鼠标右键，选择“压缩”选项，如图 2.6.3.1 所示：



图 2.6.3.1 文件压缩

在图 2.6.3.1 中我们要对 test2 这个文件夹进行压缩，点击“压缩”以后会弹出图 2.6.3.2 所

示界面让选择压缩后的文件名和压缩格式。

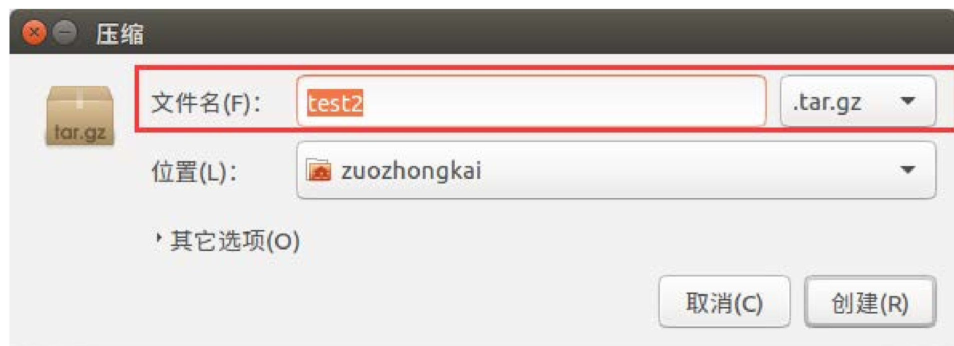


图 2.6.3.2 压缩命名和格式选择

在图 2.6.3.2 中，设置好压缩以后的文件名，然后选择压缩格式，可选的压缩格式如图 2.6.3.3 所示：

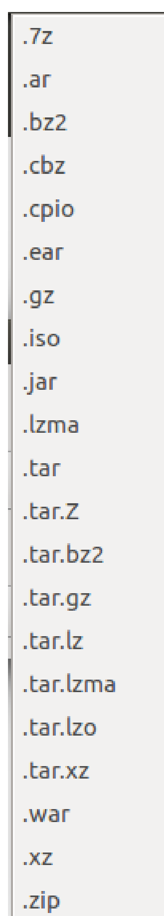


图 2.6.3.3 可选压缩格式

从图 2.6.3.3 中可以看出，可以选择的压缩格式还是有很多的，挑选一个格式进行压缩，比如我选择的“.zip”这个格式，压缩完成以后如图 2.6.3.4 所示：

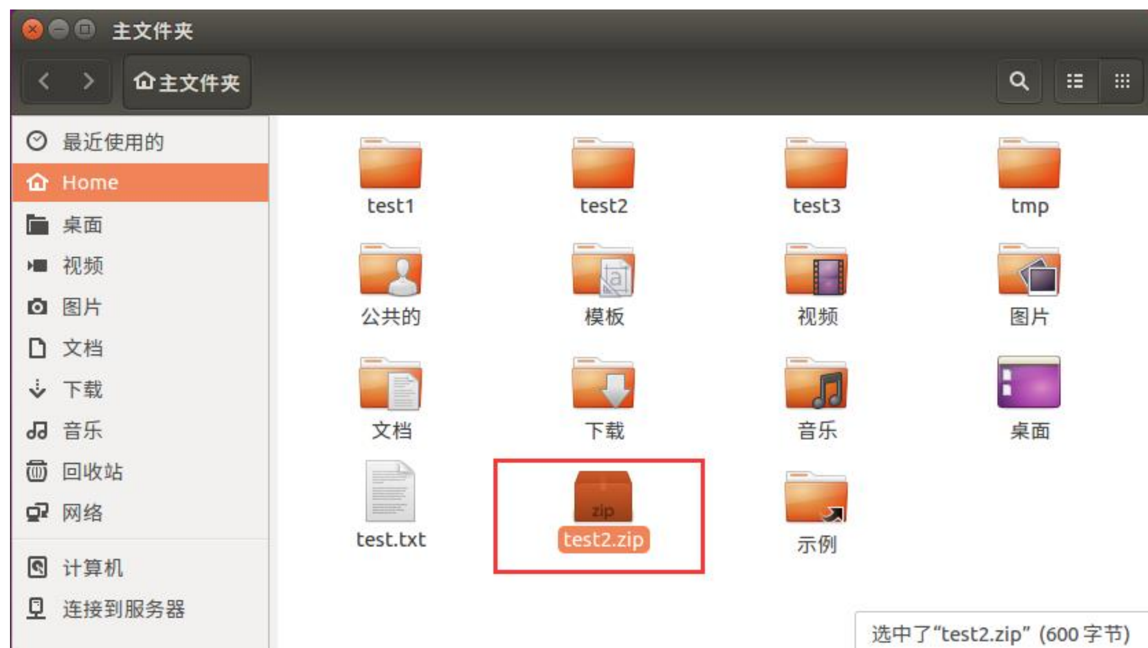


图 2.6.3.4 压缩完成的文件

上面就是使用图形化进行文件压缩的过程，我们接下来对刚刚压缩的 test2.zip 进行解压缩，鼠标放到 test2.zip 上然后点击鼠标右键，选择“提取到此处”，如图 2.6.3.5 所示：



图 2.6.3.5 解压缩文件

点击图 2.6.3.5 中的“提取到此处”以后，系统就会自动进行解压缩，上面就是在 Ubuntu 中使用图形化工具进行文件的压缩和解压缩。

2、命令行进行文件的压缩和解压缩

上面我们学习了如何使用图形化工具在 Ubuntu 下进行文件的压缩和解压缩，本节我们学学如何使用命令行进行压缩和解压缩，我们后面的开发中所有涉及到压缩和解压缩的操作都是

在命令行下完成的。命令行下进行压缩和解压缩常用的命令有三个：zip、unzip 和 tar，我们依次来学习：

、命令 zip

zip 命令看名字就知道是针对.zip 文件的，用于将一个或者多个文件压缩成一个.zip 结尾的文件，命令格式如下：

```
zip  [参数]   [压缩文件名.zip]   [被压缩的文件]
```

主要参数函数如下：

-b<工作目录> 指定暂时存放文件的目录。

-d 从 zip 文件中删除一个文件。

-F 尝试修复已经损毁的压缩文件。

-g 将文件压缩入现有的压缩文件中，不需要新建压缩文件。

-h 帮助。

-j 只保存文件的名，不保存目录。

-m 压缩完成以后删除源文件。

-n<字尾符号> 不压缩特定扩展名的文件。

-q 不显示压缩命令执行过程。

-r 递归压缩，将指定目录下的所有文件和子目录一起压缩。

-v 显示指令执行过程。

-num 压缩率，为 1~9 的数值。

上面讲解了如何使用图形化压缩工具对文件夹 test2 进行压缩，这里我们使用命令“zip”对 test2 文件夹进行压缩，操作如图 2.6.3.6 所示：

```
zuozhongkai@ubuntu:~$ ls  //显示当前目录下所有文件
examples.desktop  test2  test.txt  公共的  视频  文档  音乐
test1             test3  tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ zip -rv test2.zip test2  //使用zip命令进行压缩
  adding: test2/        (in=0) (out=0) (stored 0%)
  adding: test2/d.c     (in=0) (out=0) (stored 0%)
  adding: test2/b.c     (in=0) (out=0) (stored 0%)
  adding: test2/a.c     (in=0) (out=0) (stored 0%)
total bytes=0, compressed=0 -> 0% savings
zuozhongkai@ubuntu:~$ ls                     //显示压缩以后的当前目录下所有文件
examples.desktop  test2      test3     tmp     模板  图片  下载  桌面
test1             test2.zip  test.txt  公共的  视频  文档  音乐
```

图 2.6.3.6 使用 zip 进行文件压缩

图 2.6.3.6 就是使用 zip 命令进行 test2 文件夹的压缩，我们使用的命令如下：

```
zip  -rv  test2.zip  test2
```

上述命令中，-rv 表示递归压缩并且显示压缩命令执行过程。

命令 unzip

unzip 命令用于对.zip 格式的压缩包进行解压，命令格式如下：

```
unzip    [参数]   [压缩文件名.zip]
```

主要参数如下：

-l 显示压缩文件内所包含的文件。

-t 检查压缩文件是否损坏，但不解压。

-v 显示命令显示的执行过程。

-Z 只显示压缩文件的注解。

-C 压缩文件中的文件名称区分大小写。

-j 不处理压缩文件中的原有目录路径。

-L 将压缩文件中的全部文件名改为小写。

-n 解压缩时不要覆盖原有文件。

-P<密码> 解压密码。

-q 静默执行，不显示任何信息。

-x<文件列表> 指定不要处理.zip 中的哪些文件。

-d<目录> 把压缩文件解到指定目录下。

对上面压缩的 test2.zip 文件使用 unzip 命令进行解压缩，操作如图 2.6.3.7 所示：

```
zuozhongkai@ubuntu:~$ rm -rf test2 //删除以前的test2文件夹，防止干扰解压过程
zuozhongkai@ubuntu:~$ ls          //显示当前目录下所有文件
examples.desktop  test2.zip  test.txt  公共的  视频  文档  音乐
test1             test3      tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ unzip test2.zip  //解压缩test2.zip文件
Archive:  test2.zip
   creating: test2/
  extracting: test2/d.c
  extracting: test2/b.c
  extracting: test2/a.c
zuozhongkai@ubuntu:~$ ls          //显示当前目录下所有文件
examples.desktop  test2      test3     tmp     模板  图片  下载  桌面
test1             test2.zip  test.txt  公共的  视频  文档  音乐
```

图 2.6.3.7 命令 unzip 使用演示

、命令 tar

我们前面讲的 zip 和 unzip 这两个是命令只适用于.zip 格式的压缩和解压，其它压缩格式就用不了了，比如 Linux 下最常用的.bz2 和.gz 这两种压缩格式。其它格式的压缩和解压使用命令 tar，tar 将压缩和解压缩集合在一起，使用不同的参数即可，命令格式如下：

```
tar  [参数]  [压缩文件名]   [被压缩文件名]
```

常用参数如下：

-c 创建新的压缩文件。

-C<目的目录> 切换到指定的目录。

-f<备份文件> 指定压缩文件。

-j 用 tar 生成压缩文件，然后用 bzip2 进行压缩。

-k 解开备份文件时，不覆盖已有的文件。

-m 还原文件时，不变更文件的更改时间。

-r 新增文件到已存在的备份文件的结尾部分。

-t 列出备份文件内容。

-v 显示指令执行过程。

-w 遭遇问题时先询问用户。

-x 从备份文件中释放文件，也就是解压缩文件。

-z 用 tar 生成压缩文件，用 gzip 压缩。

-Z 用 tar 生成压缩文件，用 compress 压缩。

使用 tar 命令来进行.zip 和.gz 格式的文件压缩，操作如图 2.6.3.8 所示：

```
zuozhongkai@ubuntu:~$ ls    //显示当前目录下的所有文件
examples.desktop  test2  test.txt  公共的  视频  文档  音乐
test1             test3  tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ tar -vcjf test1.tar.bz2 test1 //压缩为.bz2格式
test1/
test1/c.c
test1/b.c
test1/a.c
zuozhongkai@ubuntu:~$ ls                      //查看压缩后的文件是否存在
examples.desktop  test1.tar.bz2  test3     tmp     模板  图片  下载  桌面
test1             test2          test.txt  公共的  视频  文档  音乐
zuozhongkai@ubuntu:~$ tar -vczf test1.tar.gz test1 //压缩为.gz格式
test1/
test1/c.c
test1/b.c
test1/a.c
zuozhongkai@ubuntu:~$ ls                      //查看压缩后的文件是否存在
examples.desktop  test1.tar.bz2  test2  test.txt  公共的  视频  文档  音乐
test1             test1.tar.gz   test3  tmp       模板    图片  下载  桌面
```

图 2.6.3.8 tar 命令进行压缩

在图 2.6.3.8 中，我们使用如下两个命令将 test1 文件夹压缩为.bz2 和.gz 这两个格式：

```
tar -vcjf test1.tar.bz2 test1
tar -vczf test1.tar.gz test1
```

在上面两行命令中，-vcjf 表示创建 bz2 格式的压缩文件，-vczf 表示创建.gz 格式的压缩文件。学习了如何使用 tar 命令来完成压缩，我们再来学习使用 tar 命令完成文件的解压，操作如图 2.6.3.9 所示：

```
zuozhongkai@ubuntu:~$ ls        //显示当前目录下所有文件
examples.desktop  test2.tar.gz  test.txt  公共的  视频  文档  音乐
test1.tar.bz2     test3         tmp       模板    图片  下载  桌面
zuozhongkai@ubuntu:~$ tar -vxjf test1.tar.bz2  //解压缩.bz2格式文件
test1/
test1/c.c
test1/b.c
test1/a.c
zuozhongkai@ubuntu:~$ ls               //检查test1.tar.bz2是否解压缩成功
examples.desktop  test1.tar.bz2  test3     tmp     模板  图片  下载  桌面
test1             test2.tar.gz   test.txt  公共的  视频  文档  音乐
zuozhongkai@ubuntu:~$ tar -vxzf test2.tar.gz  //解压缩.gz格式文件
test2/
test2/d.c
test2/b.c
test2/a.c
                                       //检查test2.tar.gz是否解压缩成功
zuozhongkai@ubuntu:~$ ls
examples.desktop  test2         test.txt  模板  文档  桌面
test1             test2.tar.gz  tmp       视频  下载
test1.tar.bz2     test3         公共的    图片  音乐
```

图 2.6.3.9 tar 解压缩命令

图 2.6.3.9 中我们使用如下所示两行命令完成.bz2 和.gz 格式文件的解压缩：

```
tar -vxjf test1.tar.bz2
tar -vxzf test2.tar.gz
```

上述两行命令中，-vxjf 用来完成.bz2 格式压缩文件的解压，-vxzf 用来完成.gz 格式压缩文件的解压。关于 Ubuntu 下的命令行压缩和解压缩就讲解到这里，重点是 tar 命令，要熟练掌握使用 tar 命令来完成.bz2 和.gz 格式的文件压缩和解压缩。

2.6.4 文件查询和搜索

文件的查询和搜索也是最常用的操作，在嵌入式 Linux 开发中常常需要在 Linux 源码文件中查询某个文件是否存在，或者搜索哪些文件都调用了某个函数等等。本节我们就讲解两个最常用的文件查询和搜索命令：find 和 grep。

1、命令 find

find 命令用于在目录结构中查找文件，其命令格式如下：

```
find [路径] [参数] [关键字]
```

路径是要查找的目录路径，如果不写的话表示在当前目录下查找，关键字是文件名的一部分，主要参数如下：

-name<filename> 按照文件名称查找，查找与 filename 匹配的文件，可使用通配符。

-depth 从指定目录下的最深层的子目录开始查找。

-gid<群组识别码> 查找符合指定的群组识别码的文件或目录。

-group<群组名称> 查找符合指定的群组名称的文件或目录。

-size<文件大小> 查找符合指定文件大小的文件。

-type<文件类型> 查找符合指定文件类型的文件。

-user<拥有者名称> 查找符合指定的拥有者名称的文件或目录。

find 命令的参数有很多，常用的就这些，关于其它的参数大家可以自行上网查找，我们来看一下如何使用 find 命令进行文件搜索，我们搜索目录/etc 中以“vim”开头的文件为例，操作如图 2.6.4.1 所示：

```
zuozhongkai@ubuntu:~$ find /etc/ -name vim*
/etc/vim
/etc/vim/vimrc
/etc/vim/vimrc.tiny
find: `/etc/cups/ssl': 权限不够
/etc/alternatives/vim
/etc/alternatives/vimdiff
find: `/etc/polkit-1/localauthority': 权限不够
find: `/etc/ssl/private': 权限不够
```

图 2.6.4.1 find 命令操作

从图 2.6.4.1 可以看出，在目录/etc 下，包含以“vim*”开头的文件有/etc/vim、/etc/vim/vimrc 等等，就不一一列出了。

2、命令 grep

find 命令用于在目录中搜索文件，我们有时候需要在文件中搜索一串关键字，grep 就是完成这个功能的，grep 命令用于查找包含指定关键字的文件，如果发现某个文件的内容包含所指定的关键字，grep 命令就会把包含指定关键字的这一行标记出来，grep 命令格式如下：

```
grep [参数] 关键字 文件列表
```

grep 命令一次只能查一个关键字，主要参数如下：

-b 在显示符合关键字的那一列前，标记处该列第 1 个字符的位编号。

-c 计算符合关键字的列数。

-d<进行动作> 当指定要查找的是目录而非文件时，必须使用此参数！否则 grep 指令将回报信息并停止搜索。

-i 忽略字符大小写。

-v　反转查找，只显示不匹配的行。

-r　在指定目录中递归查找。

比如我们在目录/usr 下递归查找包含字符“Ubuntu”的文件，操作如图 2.6.4.2 所示：

```
zuozhongkai@ubuntu:~$ grep -ir "Ubuntu" /usr
匹配到二进制文件 /usr/bin/fcitx-qimpanel-configtool
匹配到二进制文件 /usr/bin/nsupdate
/usr/bin/apport-bug:# Author: Martin Pitt <martin.pitt@ubuntu.com>
/usr/bin/apport-bug:# this so that confined applications using ubuntu-browsers.d
/ubuntu-integration
匹配到二进制文件 /usr/bin/locale
/usr/bin/fcitx-configtool:          LXDE|Lubuntu)
匹配到二进制文件 /usr/bin/x86_64-linux-gnu-elfedit
```

图 2.6.4.2 命令 grep 演示

2.6.5 文件类型

这里的文件类型不是说这个文件是音乐文件还是文本文件，在用户根目录下使用命令“ls -l”来查看用户根目录下所有文件的详细信息，如图 2.6.5.1 所示：

```
zuozhongkai@ubuntu:~$ ls -l
总用量 72
-rw-r--r-- 1 zuozhongkai zuozhongkai 8980 12月  18 02:08 examples.desktop
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  24 21:56 test1
-rw-rw-r-- 1 zuozhongkai zuozhongkai  194 12月  24 21:58 test1.tar.bz2
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  23 01:00 test2
-rw-rw-r-- 1 zuozhongkai zuozhongkai  171 12月  24 22:09 test2.tar.gz
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  23 00:45 test3
-rw-rw-r-- 1 zuozhongkai zuozhongkai   68 12月  21 01:40 test.txt
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  19 21:41 tmp
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 公共的
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 模板
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 视频
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 图片
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 文档
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 下载
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 音乐
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  22 02:08 桌面
```

图 2.6.5.1 文件详细信息

在图 2.6.5.1 中，每个文件的详细信息占一行，每行最前面都是一个符号就标记了当前文件类型，比如 test1 的第一个字符是“d”，test1.tar.bz2 文件第一个字符是“-”。这些字符表示的文件类型如下：

-　普通文件，一些应用程序创建的，比如文档、图片、音乐等等。

d　目录文件。

c　字符设备文件，Linux 驱动里面的字符设备驱动，比如串口设备，音频设备等。

b　块设备文件，存储设备驱动，比如硬盘，U 盘等。

l　符号连接文件，相当于 Windwos 下的快捷方式。

s　套接字文件。

p　管道文件，主要指 FIFO 文件。

我们后面学习 Linux 驱动开发的时候基本是在和字符设备文件和块设备文件打交道。

2.7 Linux 用户权限管理

2.7.1　Ubuntu 用户系统

Ubuntu 是一个多用户系统，我们可以给不同的使用者创建不同的用户账号，每个用户使用各自的账号登陆，使用用户账号的目的一是方便系统管理员管理，控制不同用户对系统的访问权限，另一方面是为用户提供安全性保护。

我们前面在安装 Ubuntu 系统的时候被要求创建一个账户，当我们创建好账号以后，系统会在目录/home 下以该用户名创建一个文件夹，所有与该用户有关的文件都会被存储在这个文件文件夹中。同样的，创建其它用户账号的时候也会在目录/home 下生成一个文件夹来存储该用户的文件，图 2.7.1.1 就是我的电脑上“zuozhongkai”这个账户的文件夹。

```
zuozhongkai@ubuntu:~$ ls
examples.desktop  tmp  公共的  模板  视频  图片  文档  下载  音乐  桌面
```

图 2.7.1.1 用户账号根目录

装系统的时候创建的用户其权限比后面创建的用户大一点，但是没有 root 用户权限大，Ubuntu 下用户类型分为以下 3 类：

- 初次创建的用户，此用户可以完成比普通用户更多的功能。
- root 用户，系统管理员，系统中的玉皇大帝，拥有至高无上的权利。
- 普通用户，安装完操作系统以后被创建的用户。

以上三种用户，每个用户都有一个 ID 号，称为 UID，操作系统通过 UID 来识别是哪个用户，用户相关信息可以在文件/etc/passwd 中查看到，如图 2.7.1.2 所示：

```
zuozhongkai@ubuntu:/$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/bin/false
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
zuozhongkai:x:1000:1000:zuozhongkai,,,:/home/zuozhongkai:/bin/bash
guest-sjhdig:x:999:999:访客:/tmp/guest-sjhdig:/bin/bash
```

图 2.7.1.2 passwd 文件内容

从配置文件 passwd 中可以看到，每个用户名后面都有两个数字，比如用户“zuozhongkai”后面“1000:1000”，第一个数字是用户的 ID，另一个是用户的 GID，也就是用户组 ID。Ubuntu 里面每个用户都属于一个用户组里面，用户组就是一组有相同属性的用户集合。

2.7.2 权限管理

在使用 Windows 的时候我们很少接触到用户权限，最多就是打开某个软件出问题的时候会选择以“管理员身份”打开。Ubuntu 下我们会常跟用户权限打交道，权限就是用户对于系统资

源的使用限制情况，root 用户拥有最大的权限，可以为所欲为，装系统的时候创建的用户拥有 root 用户的部分权限，其它普通用户的权限最低。对于我们做嵌入式开发的人一般不关注用户的权限问题，因为嵌入式基本是单用户，做嵌入式开发重点关注的是文件的权限问题。

对于一个文件通常有三种权限：读(r)、写(w)和执行(x)，使用命令“ls -l”可以查看某个目录下所有文件的权限信息，如图 2.7.2.1 所示：

```
zuozhongkai@ubuntu:~$ ls -l
总用量 48
-rw-r--r-- 1 zuozhongkai zuozhongkai 8980 12月  18 02:08 examples.desktop
-rw-rw-r-- 1 zuozhongkai zuozhongkai    0 12月  25 20:44 test.c
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  19 21:41 tmp
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 公共的
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 模板
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 视频
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 图片
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 文档
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 下载
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 音乐
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  22 02:08 桌面
```

图 2.7.2.1 文件权限信息

在图 2.7.2.1 中我们以文件 test.c 为例讲解，文件 test.c 文件信息如下：

```
-rw-rw-r-- 1 zuozhongkai zuozhongkai      0 12 月  25 20:44 test.c
```

其中“-rw-rw-r--”表示文件权限与用户和用户组之间的关系，第一位表示文件类型，上一小节已经说了。剩下的 9 位以 3 位为一组，分别表示文件拥有者的权限，文件拥有者所在用户组的权限以及其它用户权限。后面的“zuozhongkai zuozhongkai”分别代表文件拥有者(用户)和该用户所在的用户组，因此文件 test.c 的权限情况如下：

、文件 test.c 的拥有者是用户 zuozhongkai，其对文件 tesst.c 的权限是“rw-”，也就是对该文件拥有读和写两种权限。

、用户 zuozhongkai 所在的用户组也叫做 zuozhongkai，其组内用户对于文件 test.c 的权限是“rw-”，也是拥有读和写这两种权限。

、其它用户对于文件 test.c 的权限是“r--”，也就是只读权限。

对于文件，可读权限表示可以打开查看文件内容，可写权限表示可以对文件进行修改，可执行权限就是可以运行此文件(如果是软件的话)。对于文件夹，拥有可读权限才可以使用命令 ls 查看文件夹中的内容，拥有可执行权限才能进入到文件夹内部。

如果某个用户对某个文件不具有相应的权限的话就不能进行相应的操作，比如根目录“/”下的文件只有 root 用户才有权限进行修改，如果以普通用户去修改的话就会提示没有权限。比如我们要在根目录“/” 创建一个文件 mytest，使用命令“touch mytest”，结果如图 2.7.2.2 所示：

```
zuozhongkai@ubuntu:/$ touch mytest
touch: 无法创建'mytest': 权限不够
```

图 2.7.2.2 创建文件

在图 2.7.2.2 中，我以用户“zuozhongkai”在根目录“/”创建文件 mytest，结果提示我无法创建“mytest”，因为权限不够，因为只有 root 用户才能在根目录“/”下创建文件。我们可以使用命令“sudo”命令暂时切换到 root 用户，这样就可以在根目录“/”下创建文件 mytest 了，如图 2.7.2.3 所示：

```
zuozhongkai@ubuntu:/$ sudo touch mytest
[sudo] zuozhongkai 的密码:
zuozhongkai@ubuntu:/$ ls
bin    dev   initrd.img      lib64       mnt     proc  sbin  sys  var
boot   etc   initrd.img.old  lost+found  mytest  root  snap  tmp  vmlinuz
cdrom  home  lib             media       opt     run   srv   usr  vmlinuz.old
```

图 2.7.2.3 使用 sudo 命令创建文件

在图 2.7.2.3 中我们使用命令"sduo"以后就可以在根目录"/"创建文件 mytest，在进行其它的操作的时候，遇到提示权限不够的时候都可以使用 sudo 命令暂时以 root 用户身份去执行。

上面我们讲了，文件的权限有三种：读(r)、写(w)和执行(x)，除了用 r、w 和 x 表示以外，我们也可以使用二进制数表示，三种权限就可以使用 3 位二进制数来表示，一种权限对应一个二进制位，如果该位为 1 就表示具备此权限，如果该位为 0 就表示没不具备此权限，如表 2.7.2.1 所示：

字母	二进制	八进制
r	100	4
w	010	2
x	001	1

表 2.7.2.1 文件权限数字表示方法

如果做过单片机开发的话，就会发现和单片机里面的寄存器位一样，将三种权限 r、w 和 x 进行不同的组合，即可得到不同的二进制数和八进制数，3 位权限可以组出 8 种不同的权限组合，如表 2.7.2.2 所示：

权限	二进制数字	八进制数字
---	000	0
--x	001	1
-w-	010	2
-wx	011	3
r--	100	4
r-x	101	5
rw-	110	6
rwx	111	7

表 2.7.2.2 文件所有权限组合

表 2.7.2.2 中权限所对应的八进制数字就是每个权限对应的位相加，比如权限 rwx 就是 4+2+1=7。前面的文件 test.c 其权限为"rw-rw-r--"，因此其十进制表示就是：664。

另外我们也开始使用 a、u、g 和 o 表示文件的归属关系，用=、+和-表示文件权限的变化，如表 2.7.2.3 所示：

字母	意义
r	可读权限
w	可写权限
x	可执行权限
a	所有用户
u	归属用户
g	归属组
o	其它用户

=	具备权限
+	添加某权限
-	去除某权限

表 2.7.2.3 权限修改字母表示方式

对于文件 test.c，我们想要修改其归属用户(zuozhongkai)对其拥有可执行权限，那么就可以使用：u+x。如果希望设置归属用户及其所在的用户组都对其拥有可执行权限就可以使用：gu+x。

2.7.3 权限管理命令

我们也可以使用 Shell 来操作文件的权限管理，主要用到“chmod”和“chown”这两个命令，我们一个一个来看。

1、权限修改命令 chmod

命令“chmod”用于修改文件或者文件夹的权限，权限可以使用前面讲的数字表示也可以使用字母表示，命令格式如下：

```
chmod    [参数]    [文件名/目录名]
```

主要参数如下：

-c 效果类似“-v”参数，但仅回显更改的部分。

-f 不显示错误信息。

-R 递归处理，指定目录下的所有文件及其子文件目录一起处理。

-v 显示指令的执行过程。

我们先来学习以下如何使用命令“chmod”修改一个文件的权限，在用户根目录下创建一个文件 test，然后查看其默认权限，操作如图 2.7.3.1 所示：

```
zuozhongkai@ubuntu:~$ touch test
zuozhongkai@ubuntu:~$ ls -l
总用量 48
-rw-r--r-- 1 zuozhongkai zuozhongkai 8980 12月  18 02:08 examples.desktop
-rw-rw-r-- 1 zuozhongkai zuozhongkai    0 12月  25 22:24 test
drwxrwxr-x 2 zuozhongkai zuozhongkai 4096 12月  19 21:41 tmp
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 公共的
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 模板
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 视频
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 图片
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 文档
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 下载
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  18 02:21 音乐
drwxr-xr-x 2 zuozhongkai zuozhongkai 4096 12月  22 02:08 桌面
```

图 2.7.3.1 创建文件 test

在图 2.7.3.1 中我们创建了一个文件：test，这个文件的默认权限为“rw-rw-r--”，我们将其权限改为“rwxrw-rw”，对应数字就是 766，操作如下：

```
zuozhongkai@ubuntu:~$ ls -l test    //查看文件test权限
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月  25 22:24 test
zuozhongkai@ubuntu:~$ chmod 766 test //修改文件test的权限为766
zuozhongkai@ubuntu:~$ ls -l test     //查看修改以后的文件test权限
-rwxrw-rw- 1 zuozhongkai zuozhongkai 0 12月  25 22:24 test
```

图 2.7.3.2 修改权限

在图 2.7.3.2 中，我们修改文件 test 的权限为 766，修改完成以后的 test 文件权限为“rwxrw-rw-”，和我们设置的一样，说明权限修改成功。

上面我们是通过数字来修改权限的，我们接下来使用字母来修改权限，操作如图 2.7.3.3 所示：

```
zuozhongkai@ubuntu:~$ touch a.c //创建文件a.c
zuozhongkai@ubuntu:~$ ls -l a.c //显示文件a.c的权限
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月 25 22:33 a.c
zuozhongkai@ubuntu:~$ chmod u+x a.c //给文件a.c的归属用户添加可执行权限
zuozhongkai@ubuntu:~$ ls -l a.c //显示修改权限后的文件a.c
-rwxrw-r-- 1 zuozhongkai zuozhongkai 0 12月 25 22:33 a.c
```

图 2.7.3.3 使用字母修改文件权限

上面两个例子都是修改文件的权限，接下来我们修改文件夹的权限，新建一个 test 文件夹，在文件夹 test 里面创建 a.c、b.c 和 c.c 三个文件，如图 2.7.3.4 所示：

```
zuozhongkai@ubuntu:~$ ls -l test/
总用量 0
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月 26 00:20 a.c
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月 26 00:20 b.c
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月 26 00:20 c.c
```

图 2.7.3.4 test 文件夹

在图 2.7.3.4 中 test 文件夹下的文件 a.c、b.c 和 c.c 的权限均为“rw-rw-r--”，我们将 test 文件夹下的所有文件权限都改为“rwxrwxrwx”，也就是数字 777，操作如图 2.7.3.5 所示：

```
zuozhongkai@ubuntu:~$ chmod -R 777 test/ //递归修改文件权限
zuozhongkai@ubuntu:~$ ls -l test/ //查看修改权限以后的文件
总用量 0
-rwxrwxrwx 1 zuozhongkai zuozhongkai 0 12月 26 00:20 a.c
-rwxrwxrwx 1 zuozhongkai zuozhongkai 0 12月 26 00:20 b.c
-rwxrwxrwx 1 zuozhongkai zuozhongkai 0 12月 26 00:20 c.c
```

图 2.7.3.5 递归修改文件夹权限

2、文件归属者修改命令 chown

命令 chown 用来修改某个文件或者目录的归属者用户或者用户组，命令格式如下：

```
chown    [参数]    [用户名.<组名>]    [文件名/目录]
```

其中[用户名.<组名>]表示要将文件或者目录改为哪一个用户或者用户组，用户名和组名用“.”隔开，其中用户名和组名中的任何一个都可以省略，命令主要参数如下：

-c 效果同-v 类似，但仅回报更改的部分。
-f 不显示错误信息。
-h 只对符号连接的文件做修改，不改动其它任何相关的文件。
-R 递归处理，将指定的目录下的所有文件和子目录一起处理。
-v 显示处理过程。

在用户根目录下创建一个 test 文件，查看其文件夹所属用户和用户组，如图 2.7.3.6 所示：

```
zuozhongkai@ubuntu:~$ touch test
zuozhongkai@ubuntu:~$ ls -l test
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月 26 00:36 test
```

图 2.7.3.6 test 文件信息查询

从图 2.7.3.6 中可以看出，文件 test 的归属用户为 zuozhongkai，所属的用户组为 zuozhongkai，将文件 test 归属用户改为 root 用户，所属的用户组也改为 root，操作如图 2.7.3.7 所示：

```
zuozhongkai@ubuntu:~$ ls -l test    //显示文件test的归属用户和归属组
-rw-rw-r-- 1 zuozhongkai zuozhongkai 0 12月  26 00:45 test
zuozhongkai@ubuntu:~$ sudo chown root.root test  //修改文件test的归属用户和归属组
zuozhongkai@ubuntu:~$ ls -l test   //查看修改以后的文件test归属用户和归属组
-rw-rw-r-- 1 root root 0 12月  26 00:45 test
```

图 2.7.3.7 修改文件归属用户和归属组

命令 shown 同样也可以递归处理来修改文件夹的归属用户和用户组，用法和命令 chown 一样，这里就不演示了。

2.8 Linux 磁盘管理

2.8.1 Linux 磁盘管理基本概念

Linux 的磁盘管理体系和 Windows 有很大的区别，在 Windows 下经常会遇到“分区”这个概念，在 Linux 中一般不叫“分区”而叫“挂载点”。 “挂载点”就是将一个硬盘的一部分做成文件夹的形式，这个文件夹的名字就是“挂载点”，不管在哪个发行版的 Linux 中，用户是绝对看到不到 C 盘、D 盘这样的概念的，只能看到以文件夹形式存在的“挂载点”.

文件/etc/fstab 详细的记录了 Ubuntu 中硬盘分区的情况，如图 2.8.1.1 所示：

```
zuozhongkai@ubuntu:~$ cat /etc/fstab
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda1 during installation
UUID=fcdae2a0-05f1-48fa-a6c8-b4efff06a180 /               ext4    errors=remount-ro 0       1
# swap was on /dev/sda5 during installation
UUID=c9eb5d28-50fe-4e35-951d-ea8a1c4b7810 none            swap    sw              0       0
```

图 2.8.1.1 文件 fstab

在图 2.8.1.1 中有一行“/ was on /dev/sda1 during installation”，意思是根目录“/”是在/dev/sda1 上的，其中“/”是挂载点，“/dev/sda1”就是我们装 Ubuntu 系统的硬盘。由于我们的系统是安装在虚拟机中的，因此图 2.8.1.1 没有出现实际的硬盘。可以通过如下命令查看当前系统中的磁盘：

```
ls /dev/sd*
```

上述命令就是打印出所有以/dev/sd 开头的设备文件，如图 2.8.1.2 所示：

```
zuozhongkai@ubuntu:~$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5
```

图 2.8.1.2 查看硬盘设备文件

在图 2.8.1.2 中有四个磁盘设备文件，其中 sd 表示是 SATA 硬盘或者其它外部设备，最后面的数字表示该硬盘上的第 n 个分区，比如/dev/sda1 就表示磁盘 sda 上的第一个分区。图 2.8.1.2 中都是以/dev/sda 开头的，说明当前只有一个硬盘。如果再插上 U 盘、SD 卡啥的就可能会出现/dev/sdb，/dev/sdc 等等。如果你的 U 盘有两个分区那么可能就会出现/dev/sdb1、dev/sdb2 这样的设备文件。比如我现在插入我的 U 盘，插入 U 盘会提示 U 盘是接到主机还是虚拟机，如图 2.8.1.3 所示：



图 2.8.1.3 U 盘连接选择

设置好图 2.8.1.3 以后，点击“确定”按钮 U 盘就会自动连接到虚拟机中，也就是连接到 Ubuntu 系统中，我们再次使用命令“ls /dev/sd*”来查看当前的“/dev/sd*”设备文件，如图 2.8.1.4 所示：

```
zuozhongkai@ubuntu:~$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5  /dev/sdb  /dev/sdb1
```

图 2.8.1.4 插入 U 盘后的设备文件

从图 2.8.1.4 可以看出，相比图 2.8.1.2 多了/dev/sdb 和/dev/sdb1 这两个文件，其中/dev/sdb 就是 U 盘文件，/dev/sdb1 表示 U 盘的第一个分区，因为我的 U 盘就一个分区。

2.8.2 磁盘管理命令

本节我们学习以下跟磁盘操作有关的命令，这些命令如下：

1、磁盘分区命令 fdisk

如果要对某个磁盘进行分区，可以使用命令 fdisk，命令格如下：

```
fdisk    [参数]
```

主要参数如下：

-b<分区大小> 指定每个分区的大小。

-l 列出指定设备的分区表。

-s<分区编号> 将指定的分区大小输出到标准的输出上，单位为块。

-u 搭配“-l”参数，会用分区数目取代柱面数目，来表示每个分区的起始地址。

比如我要对 U 盘进行分区，千万不要对自己装 Ubuntu 系统进行分区！！！可以使用如下命令：

```
sudo  fdisk  /dev/sdb
```

结果如图 2.8.2.1 所示：

```
zuozhongkai@ubuntu:~$ sudo fdisk /dev/sdb
[sudo] zuozhongkai 的密码：

Welcome to fdisk (util-linux 2.27.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


命令(输入 m 获取帮助)：
```

图 2.8.2.1 U 盘分区界面

在图 2.8.2.1 中提示我们输入“m”可以查看帮助，因为 fdisk 还有一些字命令，通过输入“m”可以查看都有哪些子命令，如图 2.8.2.2 所示：

```
命令(输入 m 获取帮助)： m

Help:

  DOS (MBR)
   a   toggle a bootable flag
   b   edit nested BSD disklabel
   c   toggle the dos compatibility flag

  Generic
   d   delete a partition
   F   list free unpartitioned space
   l   list known partition types
   n   add a new partition
   p   print the partition table
   t   change a partition type
   v   verify the partition table
   i   print information about a partition

  Misc
   m   print this menu
   u   change display/entry units
   x   extra functionality (experts only)

  Script
   I   load disk layout from sfdisk script file
   O   dump disk layout to sfdisk script file

  Save & Exit
   w   write table to disk and exit
   q   quit without saving changes

  Create a new label
   g   create a new empty GPT partition table
   G   create a new empty SGI (IRIX) partition table
   o   create a new empty DOS partition table
   s   create a new empty Sun partition table
```

图 2.8.2.2 fdisk 命令的子命令

图 2.8.2.2 中常用的命令如下：

p 显示现有的分区

n 建立新分区

t 更改分区类型

d 删除现有的分区

a 更改分区启动标志

w 对分区的更改写入到硬盘或者存储器中。

q 不保存退出。

由于我的 U 盘里面还有一些重要的文件，所以不能现在不能进行分区，所以现在就不演示 fdisk 的分区操作了，后面我们讲解裸机例程的时候需要将可执行的 bin 文件烧写到 SD 卡，中烧写到 SD 卡之前需要对 SD 卡进行分区，到时候在详细讲解如何使用 fdisk 命令对磁盘进行分区。

2、格式化命令 mkfs

使用命令 fdisk 创建好一个分区以后，我们需要对其格式化，也就是在这个分区上创建一个文件系统，Linux 下的格式化命令为 mkfs，命令格式如下：

```
mkfs    [参数]   [-t 文件系统类型]    [分区名称]
```

主要参数如下：

fs 指定建立文件系统时的参数

-V 显示版本信息和简要的使用方法。

-v 显示版本信息和详细的使用方法。

比如我们要格式化 U 盘的分区/dev/sdb1 为 FAT 格式，那么就可以使用如下命令：

```
mkfs  –t   vfat  /dev/sdb1
```

3、挂载分区命令 mount

我们创建好分区并且格式化以后肯定是要使用硬盘或者 U 盘的，那么如何访问磁盘呢？比如我的 U 盘就一个分区，为/dev/sdb1，如果直接打开文件/dev/sdb1 会发现根本就不是我们要的结果。我们需要将/dev/sdb1 这个分区挂载到一个文件夹中，然后通过这个文件访问 U 盘，磁盘挂载命令为 mount，命令格式如下：

```
mount   [参数]   -t   [类型]   [设备名称]   [目的文件夹]
```

命令主要参数有：

-V 显示程序版本。

-h 显示辅助信息。

-v 显示执行过程详细信息。

-o ro 只读模式挂载。

-o rw 读写模式挂载。

-s-r 等于-o ro。

-w 等于-o rw。

挂载点是一个文件夹，因此在挂载之前先要创建一个文件夹，一般我们把挂载点放到“/mnt”目录下，在“/mnt”下创建一个 tmp 文件夹，然后将 U 盘的/dev/sdb1 分区挂载到/mnt/tmp 文件夹里面，操作如图 2.8.2.3 所示：

```
zuozhongkai@ubuntu:~$ ls /mnt  //查看/mnt目录下的所有文件
zuozhongkai@ubuntu:~$
zuozhongkai@ubuntu:~$ sudo mkdir /mnt/tmp  //创建文件夹/mnt/tmp
[sudo] zuozhongkai 的密码:
zuozhongkai@ubuntu:~$ ls /mnt  //查看/mnt目录下的所有文件
tmp
zuozhongkai@ubuntu:~$ sudo mount -t vfat /dev/sdb1 /mnt/tmp //挂载U盘到/mnt/tmp中
zuozhongkai@ubuntu:~$ ls /mnt/tmp //查看/mnt/tmp中的文件，也就是U盘里面的文件
AD                                IMX6UL_ZERO_V1.0.zip      开发板光盘
ARM裸机与嵌入式Linux驱动开发V1.0.docx  System Volume Information
```

图 2.8.2.3 挂载 U 盘

在图 2.8.2.3 中我们将 U 盘以 fat 格式挂载到目录/mnt/tmp 中，然后我们就可以通过访问/mnt/tmp 来访问 U 盘了。

4、卸载命令 umount

当我们不在需要访问已经挂载的 U 盘，可以通过 umount 将其从卸载点卸除，命令格式如下：

```
umount   [参数]   -t   [文件系统类型]   [设备名称]
```

-a 卸载/etc/mtab 中的所有文件系统。

-h 显示帮助。

-n 卸载时不要将信息存入到/etc/mtab 文件中

-r 如果无法成功卸载，则尝试以只读的方式重新挂载。

-t<文件系统类型> 仅卸载选项中指定的文件系统。

-v 显示执行过程。

上面我们将 U 盘挂载到了文件夹/mnt/tmp 里面，这里我们使用命令 umount 将其卸载掉，操作如图 2.8.2.4 所示：

```
zuozhongkai@ubuntu:~$ ls /mnt/tmp //查看卸载之前的文件夹/mnt/tmp
AD                                IMX6UL_ZERO_V1.0.zip      开发板光盘
ARM裸机与嵌入式Linux驱动开发V1.0.docx  System Volume Information
zuozhongkai@ubuntu:~$ sudo umount -t vfat /dev/sdb1  //卸载U盘的/dev/sdb1分区
zuozhongkai@ubuntu:~$ ls /mnt/tmp                     //查看卸载以后的/mnt/tmp文件夹
zuozhongkai@ubuntu:~$
zuozhongkai@ubuntu:~$
```

图 2.8.2.4 卸载 U 盘

在图 2.8.2.4 中，我们使用命令 umount 卸载了 U 盘，卸载以后当我们再去访问文件夹/mnt/tmp 的时候发现里面没有任何文件了，说明我们卸载成功了。

第三章 Linux C 编程入门

在 Windows 下我们可是使用各种各样的 IDE 进行编程，比如强大的 Visual Studio。但是在 Ubuntu 下如何进行编程呢？Ubuntu 下也有一些可以进行编程的工具，但是大多都只是编辑器，也就是只能进行代码编辑，如果要编译的话就需要用到 GCC 编译器，使用 GCC 编译器肯定就要接触到 Makefile。本章就讲解如何在 Ubuntu 下进行 C 语言的编辑和编译、GCC 和 Makefile 的使用和编写。通过本章的学习可以掌握 Linux 进行 C 编程的基本方法，为以后的 ARM 裸机和 Linux 驱动学习做准备。

3.1 Hello World！

我们所说的编写代码包括两部分：代码编写和编译，在 Windows 下可以使用 Visual Studio 来完成这两部分，可以在 Visual Studio 下编写代码然后直接点击编译就可以了。但是在 Linux 下这两部分是分开的，比如我们用 VIM 进行代码编写，编写完成以后在使用 GCC 编译器进行编译，其中代码编写工具很多，比如 VIM 编辑器、Emacs 编辑器、VScode 编辑器等等，本教程使用 Ubuntu 自带的 VIM 编辑器。先来编写一个最简单的“Hello World”程序，把 Linux 下的 C 编程完整的走一遍。

3.1.1 编写代码

先在用户根目录下创建一个工作文件夹：C_Program，所有的 C 语言练习都保存到这个工作文件夹下，创建过程如图 3.1.1.1 所示：

```
zuozhongkai@ubuntu:~$ mkdir C_Program
zuozhongkai@ubuntu:~$ ls
C_Program         test    模板  图片  下载  桌面
examples.desktop  公共的  视频  文档  音乐
```

图 3.1.1.1 创建工作目录

进入图 3.1.1.1 创建的 C_Program 工作文件夹，为了方便管理，我们后面每个例程都创建一个文件夹来保存所有与本例程有关的文件，创建一个名为“3.1”的文件夹来保存我们的“Hello World”程序相关的文件，操作如图 3.1.1.2 所示：

```
zuozhongkai@ubuntu:~$ cd C_Program/
zuozhongkai@ubuntu:~/C_Program$ mkdir 3.1
zuozhongkai@ubuntu:~/C_Program$ ls
3.1
zuozhongkai@ubuntu:~/C_Program$
```

图 3.1.1.2 创建工程文件夹

前面说了我们使用 VI 编辑器，在使用 VI 编辑器之前我们先做如下设置：

1、设置 TAB 键为 4 字节

VI 编辑器默认 TAB 键为 8 空格，我们改成 4 空格，用 vi 打开文件/etc/vim/vimrc，在此文件最后面输入如下代码：

```
set ts=4
```

添加完成如图 3.1.1.3 所示：

```
" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif

set ts=4
```

图 3.1.1.3 设置 TAB 为四个空格

修改完成以后保存并关闭文件。

2、VIM 编辑器显示行号

VIM 编辑器默认是不显示行号的，不显示行号不利于代码查看，我们设置 VIM 编辑器显示行号，同样是通过在文件/etc/vim/vimrc 中添加代码来实现，在文件最后面加入下面一行代码

即可：

```
set nu
```

添加完成以后的/etc/vim/vimrc 文件如图 3.1.1.4 所示：

```
" Source a global configuration file if available
if filereadable("/etc/vim/vimrc.local")
  source /etc/vim/vimrc.local
endif

set ts=4

set nu
```

图 3.1.1.4 设置 VIM 编辑器显示行号

VIM 编辑器可以自行定制，网上有很多的博客讲解如何设置 VIM，感兴趣的可以上网看一下。设置好 VIM 编辑器以后就可以正式开始编写代码了，进入前面创建的“3.1”这个工程文件夹里面，使用 vi 指令创建一个名为“main.c”的文件，然后在里面输入如下代码：

示例代码 3.1.1.1 main.c 文件代码

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5   printf("Hello World!\n");
6 }
```

编写完成以后保存退出 vi 编辑器，可以使用“cat”命令查看代码是否编写成功，如图 3.1.1.5 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.1$ cat main.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello World!\n");
}
```

图 3.1.1.5 查阅程序源码

从图 3.1.1.5 可以看出 main.c 文件是编辑成功的，代码编辑成功以后我们需要对其进行编译。

3.1.2 编译代码

Ubuntu 下的 C 语言编译器是 GCC，GCC 编译器在我们 Ubuntu 的时候就已经默认安装好了，可以通过如下命令查看 GCC 编译器的版本号：

```
gcc -v
```

在终端中输入上述命令以后终端输出如图 3.1.2.1 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.1$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 5.4.0-6ubuntu1~16.04.11' --with-bugurl=file:///usr/shar
/doc/gcc-5/README.Bugs --enable-languages=c,ada,c++,java,go,d,fortran,objc,obj-c++ --prefix=/usr --program-suffix=-5 -
enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdi
=/usr/lib --enable-nls --with-sysroot=/ --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --wi
h-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-libmpx --enable-plugin --with-
ystem-zlib --disable-browser-plugin --enable-java-awt=gtk --enable-gtk-cairo --with-java-home=/usr/lib/jvm/java-1.5.0-
cj-5-amd64/jre --enable-java-home --with-jvm-root-dir=/usr/lib/jvm/java-1.5.0-gcj-5-amd64 --with-jvm-jar-dir=/usr/lib/
vm-exports/java-1.5.0-gcj-5-amd64 --with-arch-directory=amd64 --with-ecj-jar=/usr/share/java/eclipse-ecj.jar --enable-
bjc-gc --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enab
e-multilib --with-tune=generic --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86
64-linux-gnu
Thread model: posix
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.11)
```

图 3.1.2.1 gcc 版本查询

如果输入命令“gcc -v”命令以后，你的终端输出类似图 3.1.2.1 中的信息，那么说明你的电脑已经有 GCC 编译器了。最后下面的“gcc version 5.4.0”说明本机的 GCC 编译器版本为 5.4.0 的。注意观察在图 3.1.2.1 中有“Target: x86_64-linux-gnu”一行，这说明 Ubuntu 自带的 GCC 编译器是针对 X86 架构的，因此只能编译在 X86 架构 CPU 上运行的程序。如果想要编译在 ARM 上运行的程序就需要针对 ARM 的 GCC 编译器，也就是交叉编译器！我们是 ARM 开发，因此肯定要安装针对 ARM 架构的 GCC 交叉编译器，当然了，这是后面的事，现在我们不用管这些，只要知道不同的目标架构，其 GCC 编译器是不同的。

如何使用 GCC 编译器来编译 main.c 文件呢？GCC 编译器是命令模式的，因此需要输入命令来使用 gcc 编译器来编译文件，输入如下命令：

```
gcc main.c
```

上述命令的功能就是使用 gcc 编译器来编译 main.c 这个 c 文件，过程如图 3.1.2.2 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.1$ gcc main.c
zuozhongkai@ubuntu:~/C_Program/3.1$ ls
a.out  main.c
```

图 3.1.2.2 编译 main.c 文件

在图 3.1.2.2 中可以看到，当编译完成以后会生成一个 a.out 文件，这个 a.out 就是编译生成发的可执行文件，执行此文件看看是否和我们代码的功能一样，执行的方法很简单使用命令：“./+可执行文件”，比如本例程就是命令：./a.out，操作如图 3.1.2.3 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.1$ ls
a.out  main.c
zuozhongkai@ubuntu:~/C_Program/3.1$ ./a.out
Hello World!
```

图 3.1.2.3 执行编译得到的文件

在图 3.1.2.3 中执行 a.out 文件以后终端输出了“Hello World!”，这正是 main.c 要实现的功能，说明我们的程序没有错误。a.out 这个文件的命名是 GCC 编译器自动命名的，那我们能不能决定编译完生成的可执行文件名字呢？肯定可以的，在使用 gcc 命令的时候加上-o 来指定生成的可执行文件名字，比如编译 main.c 以后生成名为“main”的可执行文件，操作如图 3.1.2.4 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.1$ ls
a.out  main.c
zuozhongkai@ubuntu:~/C_Program/3.1$ gcc main.c -o main //指定编译生成的可执行文件名字为main
zuozhongkai@ubuntu:~/C_Program/3.1$ ls
a.out  main  main.c
zuozhongkai@ubuntu:~/C_Program/3.1$ ./main
Hello World!
```

图 3.1.2.4 指定可执行文件名字

在图 3.1.2.4 中，我们使用“gcc main.c –o main”来编译 main.c 文件，使用参数“-o”来指定

编译生成的可执行文件名字，至此我们就完成 Linux 下 C 编程和编译的一整套过程。

3.2 GCC 编译器

3.2.1 gcc 命令

在上一小节我们已经使用过 GCC 编译器来编译 C 文件了，我们使用到是 gcc 命令，gcc 命令格式如下：

```
gcc [选项]  [文件名字]
```

主要选项如下：

-c 只编译不链接为可执行文件，编译器将输入的.c 文件编译为.o 的目标文件。

-o<输出文件名> 用来指定编译结束以后的输出文件名，如果使用这个选项的话 GCC 默认编译出来的可执行文件名字为 a.out。

-g 添加调试信息，如果要使用调试工具(如 GDB)的话就必须加入此选项，此选项指示编译的时候生成调试所需的符号信息。

-O 对程序进行优化编译，如果使用此选项的话整个源代码在编译、链接的的时候都会进行优化，这样产生的可执行文件执行效率就高。

-O2 比-O 更幅度更大的优化，生成的可执行效率更高，但是整个编译过程会很慢。

3.2.2 编译错误警告

在 Windows 下不管我们用啥编译器，如果程序有语法错误的话编译的时候都会指示出来，比如开发 STM32 的时候所使用的 MDK 和 IAR，我们可以根据错误信息方便的修改 bug。那 GCC 编译器有没有错误提示呢？肯定是有的，我们可以测试以下，新名为“3.2”的文件夹，使用 vi 在文件夹“3.2”中创建一个 main.c 文件，在文件里面输入如下代码：

示例代码 3.2.2.1 mian.c 文件代码

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int a, b;
6
7      a = 3;
8      b = 4
9      printf("a+b=\n", a + b);
10 }
```

在上述代码中有两处错误：

第 8 行、第一处是“b=4”少写了个一个“；”号。

第 9 行、第二处应该是 printf(“a+b=%d\n”, a + b);

我们编译以下上述代码，看看 GCC 编译器是否能够检查出错误，编译结果如图 3.2.2.1 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.2$ gcc main.c -o main
main.c: In function ‘main’:
main.c:9:5: error: expected ‘;’ before ‘printf’
     printf("a+b=", a + b);
     ^
```

图 3.2.2.1 错误提示

从图 3.2.2.1 中可以看出有一个 error，提示在 main.c 文件的第 9 行有错误，错误类型是在 printf 之前没有“；”号，这就是第一处错误，我们在“b = 4”后面加上分号，然后接着编译，结果又提示有一个错误，如图 3.2.2.2 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.2$ gcc main.c -o main
main.c: In function ‘main’:
main.c:9:12: warning: too many arguments for format [-Wformat-extra-args]
     printf("a+b=\n", a + b);
            ^
```

图 3.2.2.2 错误提示

在图 3.2.2.2 中，提示我们说文件 main.c 的第 9 行：printf(“a+b=\n”, a + b)有 error，错误是因为太多参数了，我们将其改为：

```
printf( “a+b=%d\n” , a + b);
```

修改完成以后接着重新编译一下，结果如图 3.2.2.3 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.2$ gcc main.c -o main
zuozhongkai@ubuntu:~/C_Program/3.2$ ls
main  main.c
```

图 3.2.2.3 编译成功

在图 3.2.2.3 中我们编译成功，生成了可执行文件 main，执行一下 main，看看结果和我们设计的是否一样，如图 3.2.2.4 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.2$ ./main
a+b=7
```

图 3.2.2.4 执 行结果

可以看出，GCC 编译器和其它编译器一样，不仅能够检测出错误类型，而且标记除了错误发生在哪个文件？哪一行？方便我们去修改代码。

3.2.3 编译流程

GCC 编译器的编译流程是：预处理、汇编、编译和链接。预处理就是对程序中的宏定义等相关的内容先进行前期的处理。汇编是先将 C 文件转换为汇编文件。当 C 文件转换为汇编文件以后就是文件编译了，编译过程就是将 C 源文件编译成.o 结尾的目标文件。编译生成的.o 文件不能直接执行，而是需要最后的链接，如果你的工程有很多个 c 源文件的话最终就会有很多.o 文件，将这些.o 文件链接在一起形成完整的一个可执行文件。

上一小节演示的例程都只有一个文件，而且文件非常简单，因此可以直接使用 gcc 命令生成可执行文件，并没有先将 c 文件编译成.o 文件，然后在链接在一起。

3.3 Makefile 基础

3.3.1 何为 Makefile

上一小节我们讲了如何使用 GCC 编译器在 Linux 进行 C 语言编译，通过在终端执行 gcc 命令来完成 C 文件的编译，如果我们的工程只有一两个 C 文件还好，需要输入的命令不多，当文件有几十、上百甚至上万个的时候用终端输入 GCC 命令的方法显然是不现实的。如果我们能够编写一个文件，这个文件描述了编译哪些源码文件、如何编译那就好了，每次需要编译工程的时只需要使用这个文件就行了。这种问题怎么可能难倒聪明的程序员，为此提出了一个解决大工程编译的工具：make，描述哪些文件需要编译、哪些需要重新编译的文件就叫做 Makefile，Makefile 就跟脚本文件一样，Makefile 里面还可以执行系统命令。使用的时候只需要一个 make

命令即可完成整个工程的自动编译，极大的提高了软件开发的效率。如果大家以前一直使用 IDE 来编写 C 语言的话肯定没有听说过 Makefile 这个东西，其实这些 IDE 是有的，只不过这些 IDE 对其进行了封装，提供给大家的是已经经过封装后的图形界面了，我们在 IDE 中添加要编译的 C 文件，然后点击按钮就完成了编译。在 Linux 下用的最多的是 GCC 编译器，这是个没有 UI 的编译器，因此 Makefile 就需要我们自己来编写了。作为一个专业的程序员，是一定要懂得 Makefile 的，一是因为在 Linux 下你不得不懂 Makefile，再就是通过 Makefile 你就能了解整个工程的处理过程。

由于 Makefile 的知识比较多，完全可以单独写本书，因此本章我们只讲解 Makefile 基础入门，如果想详细的研究 Makefile，推荐大家阅读《跟我一起写 Makefile》这份文档，文档已经放到了开发板光盘->4、参考资料里面了，本章也有很多地方参考了此文档。

3.3.2 Makefile 的引入

我们完成这样一个小工程，通过键盘输入两个整形数字，然后计算他们的和并将结果显示在屏幕上，在这个工程中我们有 main.c、input.c 和 calcu.c 这三个 C 文件和 input.h、calcu.h 这两个头文件。其中 main.c 是主体，input.c 负责接收从键盘输入的数值，calcu.h 进行任意两个数相加，其中 main.c 文件内容如下：

示例代码 3.3.2.1 main.c 文件代码

```
1 #include <stdio.h>
2 #include "input.h"
3 #include "calcu.h"
4
5 int main(int argc, char *argv[])
6 {
7     int a, b, num;
8
9     input_int(&a, &b);
10    num  = calcu(a, b);
11    printf("%d + %d = %d\r\n", a, b, num);
12 }
```

input.c 文件内容如下：

示例代码 3.3.2.2 input.c 文件代码

```
1 #include <stdio.h>
2 #include "input.h"
3
4 void input_int(int *a, int *b)
5 {
6     printf("input two num:");
7     scanf("%d %d", a, b);
8     printf("\r\n");
9 }
```

calcu.c 文件内容如下：

示例代码 3.3.2.3 calcu.c 文件代码

```
1 #include "calcu.h"
```

```
2
3 int calcu(int a, int b)
4 {
5     return (a + b);
6 }
```

文件 input.h 内容如下：

示例代码 3.3.2.4 input.h 文件代码

```
1 #ifndef _INPUT_H
2 #define _INPUT_H
3
4 void input_int(int *a, int *b);
5 #endif
```

文件 calcu.h 内容如下：

示例代码 3.3.2.5 calcu.h 文件代码

```
1 #ifndef _CALCU_H
2 #define _CALCU_H
3
4 int calcu(int a, int b);
5 #endif
```

以上就是我们这个小工程的所有源文件，我们接下来使用 3.1 节讲的方法来对其进行编译，在终端输入如下命令：

```
gcc main.c calcu.c input.c -o main
```

上面命令的意思就是使用 gcc 编译器对 main.c、calcu.c 和 input.c 这三个文件进行编译，编译生成的可执行文件叫做 main。编译完成以后执行 main 这个程序，测试一下软件是否工作正常，结果如图 3.3.2.1 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.3$ ls
calcu.c  calcu.h  input.c  input.h  main  main.c
zuozhongkai@ubuntu:~/C_Program/3.3$ ./main
input two num:5 6

5 + 6 = 11
```

图 3.3.2.1 程序测试

可以看出我们的代码按照我们所设想的工作了，使用命令“gcc main.c calcu.c input.c -o main”看起来很简单是吧，只需要一行就可以完成编译，但是我们这个工程只有三个文件啊！如果几千个文件呢？再就是如果有一个文件被修改了以，使用上面的命令编译的时候所有的文件都会重新编译，如果工程有几万个文件(Linux 源码就有这么多文件！)，想想这几万个文件编译一次所需要的时间就可怕。最好的办法肯定是哪个文件被修改了，只编译这个被修改的文件即可，其它没有修改的文件就不需要再次重新编译了，为此我们改变我们的编译方法，如果第一次编译工程，我们先将工程中的文件都编译一遍，然后后面修改了哪个文件就编译哪个文件，命令如下：

```
gcc -c main.c
gcc -c input.c
gcc -c calcu.c
gcc main.o input.o calcu.o -o main
```

上述命令前三行分别是将 main.c、input.c 和 calcu.c 编译成对应的.o 文件，所以使用了“-c”选项，“-c”选项我们上面说了，是只编译不链接。最后一行命令是将编译出来的所有.o 文件链接成可执行文件 main。假如我们现在修改了 calcu.c 这个文件，只需要将 caclue.c 这一个文件重新编译成.o 文件，然后在将所有的.o 文件链接成可执行文件即，只需要下面两条命令即可：

```
gcc -c calcu.c
gcc main.o input.o calcu.o -o main
```

但是这样就又有一个问题，如果修改的文件一多，我自己可能都不记得哪个文件修改过了，然后忘记编译，然后……，为此我们需要这样一个工具：

1、如果工程没有编译过，那么工程中的所有.c 文件都要被编译并且链接成可执行程序。

2、如果工程中只有个别 C 文件被修改了，那么只编译这些被修改的 C 文件即可。

3、如果工程的头文件被修改了，那么我们需要编译所有引用这个头文件的 C 文件，并且链接成可执行文件。

很明显，能够完成这个功能的就是 Makefile 了，在工程目录下创建名为“Makefile”的文件，文件名一定要叫做“Makefile”!!! 区分大小写的哦！如图 3.3.2.2 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.3$ ls
calcu.c  calcu.h  input.c  input.h  main.c  Makefile
```

图 3.3.2.2 Makefile 文件

在图 3.3.2.2 中 Makefile 和 C 文件是处于同一个目录的，在 Makefile 文件中输入如下代码：

示例代码 3.3.2.6 Makefile 文件代码

```
1  main: main.o input.o calcu.o
2      gcc -o main  main.o input.o calcu.o
3  main.o: main.c
4      gcc -c main.c
5  input.o: input.c
6      gcc -c input.c
7  calcu.o: calcu.c
8      gcc -c calcu.c
9
10 clean:
11     rm *.o
12     rm main
```

上述代码中所有行首需要空出来的地方一定要使用“TAB”键！不要使用空格键！这是 Makefile 的语法要求，编写好得 Makefile 如图 3.3.2.3 所示：

```
 1 main: main.o input.o calcu.o
 2     gcc -o main  main.o input.o calcu.o
 3 main.o: main.c
 4     gcc -c main.c
 5 input.o: input.c
 6     gcc -c input.c
 7 calcu.o: calcu.c
 8     gcc -c calcu.c
 9
10 clean:
11     rm *.o
12     rm main
13
```

图 3.3.2.3 Makefile 源码

Makefile 编写好以后我们就可以使用 make 命令来编译我们的工程了，直接在命令行中输入“make”即可，make 命令会在当前目录下查找是否存在“Makefile”这个文件，如果存在的话就会按照 Makefile 里面定义的编译方式进行编译，如图 3.3.2.4 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.3$ ls   //查看目录下所有文件
calcu.c  calcu.h  input.c  input.h  main.c  Makefile
zuozhongkai@ubuntu:~/C_Program/3.3$ make //make命令编译工程
gcc -c main.c
gcc -c input.c
gcc -c calcu.c
gcc -o main  main.o input.o calcu.o
zuozhongkai@ubuntu:~/C_Program/3.3$ ls   //查看编译完成后的文件夹，看看编译是否成功
calcu.c  calcu.o  input.h  main     main.o
calcu.h  input.c  input.o  main.c  Makefile
```

图 3.3.2.4 Make 编译工程

在图 3.3.2.4 中，使用命令“make”编译完成以后就会在当前工程目录下生成各种.o 和可执行文件，说明我们编译成功了。使用 make 命令编译工程的时候可能会提示如图 3.3.2.5 所示错误:

```
zuozhongkai@ubuntu:~/C_Program/3.3$ make
Makefile:2: *** missing separator。 停止。
```

图 3.3.2.5 Make 失败

图 3.3.2.5 中的错误来源一般有两点:

1、Makefile 中命令缩进没有使用 TAB 键！

2、VI/VIM 编辑器使用空格代替了 TAB 键，修改文件/etc/vim/vimrc，在文件最后面加上如下所示代码：

```
set noexpandtab
```

我们修改一下 input.c 文件源码，随便加几行空行就行了，保证 input.c 被修改过即可，修改完成以后再执行一下“make”命令重新编译一下工程，结果如图 3.3.2.6 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.3$ make
gcc -c input.c
gcc -o main  main.o input.o calcu.o
```

图 3.3.2.6 重新编译工程

从图 3.3.2.6 中可以看出因为我们修改了 input.c 这个文件，所以 input.c 和最后的可执行文件 main 重新编译了，其它没有修改过的文件就没有编译。而且我们只需要输入“make”这个命令即可，非常方便，但是 Makefile 里面的代码都是什么意思呢？这就是接下来我们要讲解的。

3.4 Makefile 语法

3.4.1 Makefile 规则格式

Makefile 里面是由一系列的规则组成的，这些规则格式如下：

```
目标…... :  依赖文件集合……
    命令 1
    命令 2
    ……
```

比如下面这条规则：

```
main :   main.o   input.o   calcu.o
         gcc   -o   main   main.o   input.o   calcu.o
```

这条规则的目标是 main，main.o、input.o 和 calcu.o 是生成 main 的依赖文件，如果要更新目标 main，就必须先更新它的所有依赖文件，如果依赖文件中的任何一个有更新，那么目标也必须更新，“更新”就是执行一遍规则中的命令列表。

命令列表中的每条命令必须以 TAB 键开始，不能使用空格！

命令列表中的每条命令必须以 TAB 键开始，不能使用空格

命令列表中的每条命令必须以 TAB 键开始，不能使用空格！

make 命令会为 Makefile 中的每个以 TAB 开始的命令创建一个 Shell 进程去执行。

了解了 Makefile 的基本运行规则以后我们再来分析一下 3.3 节中“示例代码 3.3.2.6”中的 Makefile，代码如下：

```
1  main: main.o input.o calcu.o
2      gcc -o main  main.o input.o calcu.o
3  main.o: main.c
4      gcc -c main.c
5  input.o: input.c
6      gcc -c input.c
7  calcu.o: calcu.c
8      gcc -c calcu.c
9
10 clean:
11     rm *.o
12     rm main
```

上述代码中一共有 5 条规则，1~2 行为第一条规则，3~4 行为第二条规则，5~6 行为第三条规则，7~8 行为第四条规则，10~12 为第五条规则，make 命令在执行这个 Makefile 的时候其执行步骤如下：

首先更新第一条规则中的 main，第一条规则的目标成为默认目标，只要默认目标更新了那么就认为 Makefile 的工作，完成了整个 Makefile 就是为了完成这个工作。在第一次编译的时候由于 main 还不存在，因此第一条规则会执行，第一条规则依赖于文件 main.o、input.o 和 calcu.o 这个三个.o 文件，这三个.o 文件目前还都没有，因此必须先更新这三个文件。make 会查找以这三个.o 文件为目标的规则并执行。以 main.o 为例，发现更新 main.o 的是第二条规则，因此会执行第二条规则，第二条规则里面的命令为“gcc –c main.c”，这行命令很熟悉了吧，就是不链接编译 main.c，生成 main.o，其它两个.o 文件同理。最后一个规则目标是 clean，它没有依赖文件，因此会默认为依赖文件都是最新的，所以其对应的命令不会执行，当我们想要执行 clean 的话可以直接使用命令“make clean”，执行以后就会删除当前目录下所有的.o 文件以及 main，因此 clean 的功能就是完成工程的清理，“make clean”的执行过程如图 3.4.1.1 所示：

```
zuozhongkai@ubuntu:~/C_Program/3.3$ ls
calcu.c  calcu.o  input.h  main    main.o
calcu.h  input.c  input.o  main.c  Makefile
zuozhongkai@ubuntu:~/C_Program/3.3$ make clean
rm *.o
rm main
zuozhongkai@ubuntu:~/C_Program/3.3$ ls
calcu.c  calcu.h  input.c  input.h  main.c  Makefile
```

图 3.4.1.1 make clean 执行过程

从图 3.4.1.1 可以看出，当执行“make clean”命令以后，前面编译出来的.o 和 main 可执行文件都被删除掉了，也就是完成了工程清理工作。

我们在来总结一下 Make 的执行过程：

1、make 命令会在当前目录下查找以 Makefile(makefile 其实也可以)命名的文件。

2、当找到 Makefile 文件以后就会按照 Makefile 中定义的规则去编译生成最终的目标文件。

3、当发现目标文件不存在，或者目标所依赖的文件比目标文件新(也就是最后修改时间比目标文件晚)的话就会执行后面的命令来更新目标。

这就是 make 的执行过程，make 工具就是在 Makefile 中一层一层的查找依赖关系，并执行相应的命令。编译出最终的可执行文件。Makefile 的好处就是“自动化编译”，一旦写好了 Makefile 文件，以后只需要一个 make 命令即可完成整个工程的编译，极大的提高了开发效率。把 make 和 Makefile 和做菜类似，目标都是呈现出一场盛宴，它们之间的对比关系如表 3.4.1.1 所示：

make	做菜	描述
make 工具	厨师	负责将材料加工成“美味”。
gcc 编译器	厨具	加工“美味”的工具。
Makefile	食材	加工美味所需的原材料。
最终的可执行文件	最终的菜肴	最终目的，呈现盛宴

表 3.4.1.1 make 和做菜对比

总结一下，Makefile 中规则用来描述在什么情况下使用什么命令来构建一个特定的文件，这个文件就是规则的“目标”，为了生成这个“目标”而作为材料的其它文件称为“目标”的依赖，规则的命令是用来创建或者更新目标的。

除了 Makefile 的“终极目标”所在的规则以外，其它规则的顺序在 Makefile 中是没有意义的，“终极目标”就是指在使用 make 命令的时候没有指定具体的目标时，make 默认的那个目标，它是 Makefile 文件中第一个规则的目标，如果 Makefile 中的第一个规则有多个目标，那么这些目标中的第一个目标就是 make 的“终极目标”。

3.4.2 Makefile 变量

跟 C 语言一样 Makefile 也支持变量的，先看一下前面的例子：

```
main: main.o input.o calcu.o
	gcc -o main main.o input.o calcu.o
```

上述 Makefile 语句中，main.o input.o 和 calcue.o 这三个依赖文件，我们输入了两遍，我们这个 Makefile 比较小，如果 Makefile 复杂的时候这种重复输入的工作就会非常费时间，而且非常容易输错，为了解决这个问题，Makefile 加入了变量支持。不像 C 语言中的变量有 int、char 等各种类型，Makefile 中的变量都是字符串！类似 C 语言中的宏。使用变量将上面的代码修改，修改以后如下所示：

示例代码 3.4.2.1 Makefile 变量使用

```
1 #Makefile 变量的使用
2 objects = main.o input.o calcu.o
3 main: $(objects)
4 	gcc -o main $(objects)
```

我们来分析一下“示例代码 3.4.2.1”，第 1 行是注释，Makefile 中可以写注释，注释开头要用符号“#”，不能用 C 语言中的“//”或者“/**/”！第 2 行我们定义了一个变量 objects，并且给这个变量进行了赋值，其值为字符串“main.o input.o calcu.o”，第 3 和 4 行使用到了变量 objects，Makefile 中变量的引用方法是“$(变量名)”，比如本例中的“$(objects)”就是使用变量 objects。

在“示例代码 3.4.2.1”中我们在定义变量 objects 的时候使用“=”对其进行了赋值，Makefile 变量的赋值符还有其它两个“:=”和“?=”，我们来看一下这三种赋值符的区别：

1、赋值符“=”

使用“=”在给变量的赋值的时候，不一定要用已经定义好的值，也可以使用后面定义的值，比如如下代码：

示例代码 3.4.2.1 赋值符"="使用

```
1 name = zzk
2 curname = $(name)
3 name = zuozhongkai
4
5 print:
6 	@echo curname: $(curname)
```

我们来分析一下上述代码，第 1 行定义了一个变量 name，变量值为“zzk”，第 2 行也定义了一个变量 curname，curname 的变量值引用了变量 name，按照我们 C 写语言的经验此时 curname 的值就是“zzk”。第 3 行将变量 name 的值改为了“zuozhongkai”，第 5、6 行是输出变量 curname 的值。在 Makefile 要输出一串字符的话使用“echo”，就和 C 语言中的“printf”一样，第 6 行中的“echo”前面加了个“@”符号，因为 Make 在执行的过程中会自动输出命令执行过程，在命令前面加上“@”的话就不会输出命令执行过程，大家可以测试一下不加“@”的效果。使用命令“make print”来执行上述代码，结果如图 3.4.2.1：

```
zuozhongkai@ubuntu:~/C_Program$ make print
curname: zuozhongkai
zuozhongkai@ubuntu:~/C_Program$
```

图 3.4.2.1 make 执行结果

在图 3.4.2.1 中可以看到 curname 的值不是“zzk”，竟然是“zuozhongkai”，也就是变量“name”最后一次赋值的结果，这就是赋值符“=”的神奇之处！借助另外一个变量，可以将变量的真实值推到后面去定义。也就是变量的真实值取决于它所引用的变量的最后一次有效值。

2、赋值符“:=”

在“示例代码 3.4.2.1”上来测试赋值符“:=”，修改“示例代码 3.4.2.1”中的第 2 行，将其中的“=”改为“:=”，修改完成以后的代码如下：

示例代码 3.4.2.2 ":="的使用

```
1 name = zzk
2 curname := $(name)
3 name = zuozhongkai
4
```

```
5 print:
6     @echo curname: $(curname)
```

修改完成以后重新执行一下 Makefile，结果如图 3.4.2.2 所示：

```
zuozhongkai@ubuntu:~/C_Program$ make print
curname: zzk
zuozhongkai@ubuntu:~/C_Program$
```

图 3.4.2.2 make 执行结果

从图 3.4.2.2 中可以看到此时的 curname 是 zzk，不是 zuozhongkai 了。这是因为赋值符“:=”不会使用后面定义的变量，只能使用前面已经定义好的，这就是“=”和“:=”两个的区别。

3、赋值符“?=”

“?=”是一个很有用的赋值符，比如下面这行代码：

```
curname ?= zuozhongkai
```

上述代码的意思就是，如果变量 curname 前面没有被赋值，那么此变量就是“zuozhongkai”，如果前面已经赋过值了，那么就使用前面赋的值。

4、变量追加“+=”

Makefile 中的变量是字符串，有时候我们需要给前面已经定义好的变量添加一些字符串进去，此时就要使用到符号“+=”，比如如下所示代码：

```
objects = main.o inpiut.o
objects += calcu.o
```

一开始变量 objects 的值为“main.o input.o”，后面我们给他追加了一个“calcu.o”，因此变量 objects 变成了“main.o input.o calcu.o”，这个就是变量的追加。

3.4.3 Makefile 模式规则

在 3.3.2 小节中我们编写了一个 Makefile 文件用来编译工程，这个 Makefile 的内容如下：

示例代码 3.4.3.1 Makefile 文件代码

```
1  main: main.o input.o calcu.o
2      gcc -o main  main.o input.o calcu.o
3  main.o: main.c
4      gcc -c main.c
5  input.o: input.c
6      gcc -c input.c
7  calcu.o: calcu.c
8      gcc -c calcu.c
9
10 clean:
11     rm *.o
12     rm main
```

上述 Makefile 中第 3~8 行是将对应的.c 源文件编译为.o 文件，每一个 C 文件都要写一个对应的规则，如果工程中 C 文件很多的话显然不能这么做。为此，我们可以使用 Makefile 中的模式规则，通过模式规则我们就可以使用一条规则来将所有的.c 文件编译为对应的.o 文件。

模式规则中，至少在规则的目标定义中要包涵“%”，否则就是一般规则，目标中的“%”表示对文件名的匹配，“%”表示长度任意的非空字符串，比如“%.c”就是所有的以.c 结尾的

文件，类似与通配符，a.%.c 就表示以 a.开头，以.c 结束的所有文件。

当“%”出现在目标中的时候，目标中“%”所代表的值决定了依赖中的“%”值，使用方法如下：

```
%.o : %.c
    命令
```

因此“示例代码 3.4.3.1”中的 Makefile 可以改为如下形式：

示例代码 3.4.3.2 模式规则使用

```
1  objects = main.o input.o calcu.o
2  main: $(objects)
3      gcc -o main $(objects)
4
5  %.o : %.c
6      #命令
7
8  clean:
9      rm *.o
10     rm main
```

“示例代码 3.4.3.2”中第 5、6 这两行代码替代了“示例代码 3.4.3.1”中的 3~8 行代码，修改以后的 Makefile 还不能运行，因为第 6 行的命令我们还没写呢，第 6 行的命令我们需要借助另外一种强大的变量—自动化变量。

3.4.4 Makefile 自动化变量

上面讲的模式规则中，目标和依赖都是一系列的文件，每一次对模式规则进行解析的时候都会是不同的目标和依赖文件，而命令只有一行，如何通过一行命令来从不同的依赖文件中生成对应的目标？自动化变量就是完成这个功能的！所谓自动化变量就是这种变量会把模式中所定义的一系列的文件自动的挨个取出，直至所有的符合模式的文件都取完，自动化变量只应该出现在规则的命令中，常用的自动化变量如表 3.4.4.1：

自动化变量	描述
$@	规则中的目标集合，在模式规则中，如果有多个目标的话，“$@”表示匹配模式中定义的目标集合。
$%	当目标是函数库的时候表示规则中的目标成员名，如果目标不是函数库文件，那么其值为空。
$<	依赖文件集合中的第一个文件，如果依赖文件是以模式(即“%”)定义的，那么“$<”就是符合模式的一系列的文件集合。
$?	所有比目标新的依赖目标集合，以空格分开。
$^	所有依赖文件的集合，使用空格分开，如果在依赖文件中有多个重复的文件，“$^”会去除重复的依赖文件，值保留一份。
$+	和“$^”类似，但是当依赖文件存在重复的话不会去除重复的依赖文件。
$*	这个变量表示目标模式中"%"及其之前的部分，如果目标是 test/a.test.c，目标模式为 a.%.c，那么“$*”就是 test/a.test。

表 3.4.4.1 自动化变量

表 3.4.4.1 中的 7 个自动化变量中，常用的三种：$@、$<和$^，我们使用自动化变量来完

成“示例代码 3.4.3.2”中的 Makefile，最终的完整代码如下所示：

示例代码 3.4.4.1 自动化变量

```
1  objects = main.o input.o calcu.o
2  main: $(objects)
3      gcc -o main $(objects)
4  
5  %.o : %.c
6      gcc -c $<
7  
8  clean:
9      rm *.o
10     rm main
```

上述代码代码就是修改后的完成的 Makefile，可以看出相比 3.3.2 小节中的要精简了很多，核心就在于第 5、6 这两行，第 5 行使用了模式规则，第 6 行使用了自动化变量。

3.4.5 Makefile 伪目标

Makefile 有一种特殊的目标——伪目标，一般的目标名都是要生成的文件，而伪目标不代表真正的目标名，在执行 make 命令的时候通过指定这个伪目标来执行其所在规则的定义的命令。

使用伪目标的主要是为了避免 Makefile 中定义的只执行命令的目标和工作目录下的实际文件出现名字冲突，有时候我们需要编写一个规则用来执行一些命令，但是这个规则不是用来创建文件的，比如在前面的“示例代码 3.4.4.1”中有如下代码用来完成清理工程的功能：

```
clean:
    rm *.o
    rm main
```

上述规则中并没有创建文件 clean 的命令，因此工作目录下永远都不会存在文件 clean，当我们输入“make clean”以后，后面的“rm *.o”和“rm main”总是会执行。可是如果我们“手贱”，在工作目录下创建一个名为“clean”的文件，那就不一样了，当执行“make clean”的时候，规则因为没有依赖文件，所以目标被认为是最新的，因此后面的 rm 命令也就不会执行，我们预先设想的清理工程的功能也就无法完成。为了避免这个问题，我们可以将 clean 声明为伪目标，声明方式如下：

```
.PHONY : clean
```

我们使用伪目标来更改“示例代码 3.4.4.1”，修改完成以后如下：

示例代码 3.4.5.1 伪目标

```
1  objects = main.o input.o calcu.o
2  main: $(objects)
3      gcc -o main $(objects)
4  
5  .PHONY : clean
6  
7  %.o : %.c
8      gcc -c $<
9  
```

```
10 clean:
11     rm *.o
12     rm main
```

上述代码第 5 行声明 clean 为伪目标，声明 clean 为伪目标以后不管当前目录下是否存在名为“clean”的文件，输入“make clean”的话规则后面的 rm 命令都会执行。

3.4.6 Makefile 条件判断

在 C 语言中我们通过条件判断语句来根据不同的情况来执行不同的分支，Makefile 也支持条件判断，语法有两种如下：

```
<条件关键字>
    <条件为真时执行的语句>
endif
```

以及：

```
<条件关键字>
    <条件为真时执行的语句>
else
    <条件为假时执行的语句>
endif
```

其中条件关键字有 4 个：ifeq、ifneq、ifdef 和 ifndef，这四个关键字其实分为两对、ifeq 与 ifneq、ifdef 与 ifndef，先来看一下 ifeq 和 ifneq，ifeq 用来判断是否相等，ifneq 就是判断是否不相等，ifeq 用法如下：

```
ifeq (<参数 1>,  <参数  2>)
ifeq ‘<参数 1 >’，‘ <参数 2>’
ifeq “<参数 1>”，“<参数 2>”
ifeq “<参数 1>”，‘<参数 2>’
ifeq ‘<参数 1>’，“<参数 2>”
```

上述用法中都是用来比较“参数 1”和“参数 2”是否相同，如果相同则为真，“参数 1”和“参数 2”可以为函数返回值。ifneq 的用法类似，只不过 ifneq 是用来了比较“参数 1”和“参数 2”是否不相等，如果不相等的话就为真。

ifdef 和 ifndef 的用法如下：

```
ifdef <变量名>
```

如果“变量名”的值非空，那么表示表达式为真，否则表达式为假。“变量名”同样可以是一个函数的返回值。ifndef 用法类似，但是含义用户 ifdef 相反。

3.4.7 Makefile 函数使用

Makefile 支持函数，类似 C 语言一样，Makefile 中的函数是已经定义好的，我们直接使用，不支持我们自定义函数。make 所支持的函数不多，但是绝对够我们使用了，函数的用法如下：

```
$(函数名  参数集合)
```

或者

```
${函数名  参数集合}
```

可以看出，调用函数和调用普通变量一样，使用符号“$”来标识。参数集合是函数的多个参数，参数之间以逗号“,”隔开，函数名和参数之间以“空格”分隔开，函数的调用以“$”开

头。接下来我们介绍几个常用的函数，其它的函数大家可以参考《跟我一起写 Makefile》这份文档。

1、函数 subst

函数 subst 用来完成字符串替换，调用形式如下：

```
$(subst <from>,<to>,<text>)
```

此函数的功能是将字符串<text>中的<from>内容替换为<to>，函数返回被替换以后的字符串，比如如下示例：

```
$(subst   zzk,ZZK,my name is zzk)
```

把字符串“my name is zzk”中的“zzk”替换为“ZZK”，替换完成以后的字符串为“my name is ZZK”。

2、函数 patsubst

函数 patsubst 用来完成模式字符串替换，使用方法如下：

```
$(patsubst <pattern>,<replacement>,<text>)
```

此函数查找字符串<text>中的单词是否符合模式<pattern>，如果匹配就用<replacement>来替换掉，<pattern>可以使用包括通配符“%”，表示任意长度的字符串，函数返回值就是替换后的字符串。如果<replacement>中也包涵“%”，那么<replacement>中的“%”将是<pattern>中的那个“%”所代表的字符串，比如：

```
$(patsubst %.c,%.o,a.c b.c c.c)
```

将字符串“a.c b.c c.c”中的所有符合“%.c”的字符串，替换为“%.o”，替换完成以后的字符串为“a.o b.o c.o”。

3、函数 dir

函数 dir 用来获取目录，使用方法如下：

```
$(dir <names…>)
```

此函数用来从文件名序列<names>中提取出目录部分，返回值是文件名序列<names>的目录部分，比如：

```
$(dir </src/a.c>)
```

提取文件“/src/a.c”的目录部分，也就是“/src”。

4、函数 notdir

函数 notdir 看名字就是知道去除文件中的目录部分，也就是提取文件名，用法如下：

```
$(notdir <names…>)
```

此函数用与从文件名序列<names>中提取出文件名非目录部分，比如：

```
$(notdir </src/a.c>)
```

提取文件“/src/a.c”中的非目录部分，也就是文件名“a.c”。

5、函数 foreach

foreach 函数用来完成循环，用法如下：

```
$(foreach <var>, <list>,<text>)
```

此函数的意思就是把参数<list>中的单词逐一取出来放到参数<var>中，然后再执行<text>所包含的表达式。每次<text>都会返回一个字符串，循环的过程中，<text>中所包含的每个字符串会以空格隔开，最后当整个循环结束时，<text>所返回的每个字符串所组成的整个字符串将会是函数 foreach 函数的返回值。

6、函数 wildcard

通配符“%”只能用在规则中，只有在规则中它才会展开，如果在变量定义和函数使用时，通配符不会自动展开，这个时候就要用到函数 wildcard，使用方法如下：

```
$(wildcard PATTERN…)
```

比如：

```
$(wildcard *.c)
```

上面的代码是用来获取当前目录下所有的.c 文件，类似“%”。

关于 Makefile 的相关内容就讲解到这里，本节只是对 Makefile 做了最基本的讲解，确保大家能够完成后续的学习，Makefile 还有大量的知识没有提到，有兴趣的可以自行参考《跟我一起写 Makefile》这份文档来深入学习 Makefile。

第二篇 裸机开发篇

前面几章都是 Ubuntu/Linux 的基础操作，没有涉及到开发，从本篇开始我们就开始实战操作。本篇讲解 ARM 的裸机开发，也就是不带操作系统开发，就和我们开发 STM32 一样，如果有 STM32 开发经验的话学起本篇会很容易。为什么我们要先学习裸机开发呢？

1、裸机开发是了解所使用的 CPU 最直接、最简单的方法，比如本教程使用的 I.MX6U，跟 STM32 一样，裸机开发是直接操作 CPU 的寄存器。Linux 驱动开发最终也是操作的寄存器，但是在操作寄存器之前要先编写一个符合 Linux 驱动的框架。同样一个点灯驱动，裸机可能只需要十几行代码，但是 Linux 下的驱动就需要几十行代码。

2、大部分 Linux 驱动初学者都是从 STM32 转过来的，Linux 驱动开发和 STM32 开发区别很大，比如 Linux 没有 MDK、IAR 这样的集成开发环境，需要我们自己在 Ubuntu 下搭建交叉编译环境。直接上手 Linux 驱动开发可能会因为和 STM32 巨大的开发差异而打击学习信心。

3、裸机开发是连接 Cortex-M（如 STM32）单片机和 Cortex-A(如 I.MX6U)处理器的桥梁，笔者为了帮助 STM32 开发者以最小的代价转换到 Linux 驱动开发，精心准备了十几个裸机例程，根据笔者 4 年的 STM32 开发板经验，合理的安排各个裸机例程。使用 STM32 开发方式来学习 Cortex-A(I.MX6U)，降低入门难度。通过这十几个裸机例程也可以反哺 STM32，掌握很多 MDK、IAR 这种集成开发环境没有告诉你的“干货”。

废话不多说，开干吧！！！

第四章 开发环境搭建

要进行裸机开发肯定要先搭建好开发环境，我们在开始学习 STM32 的时候肯定需要安装一堆的软件，比如 MDK、IAR、串口调试助手等等，这个就是 STM32 的开发环境搭建。同样的，要想在 Ubuntu 下进行 Cortex-A(I.MX6U)开发也需要安装一些软件，也就是网上说的开发环境搭建，环境搭建好以后我们就可以进行开发了。环境搭建分为 Ubuntu 和 Windows，因为我们最熟悉 Windows，所以代码编写、查找资料啥的肯定是在 Windows 下进行的。但是 Linux 开发又必须在 Ubuntu 下进行，所以还需要搭建 Ubuntu 下的开发环境，主要是交叉编译器的安装，本章我们就分为 Ubuntu 和 Windows，讲解这两种操作系统下的环境搭建。

4.1 Ubuntu 和 Windows 文件互传

在开发的过程中会频繁的在 Windows 和 Ubuntu 下进行文件传输，比如在 Windwos 下进行代码编写，然后将编写好的代码拿到 Ubuntu 下进行编译。Windows 和 Ubuntu 下的文件互传我们需要使用 FTP 服务，设置方法如下：

1、开启 Ubuntu 下的 FTP 服务

打开 Ubuntu 的终端窗口，然后执行如下命令来安装 FTP 服务：

```
sudo apt-get install vsftpd
```

等待软件自动安装，安装完成以后使用如下 VI 命令打开/etc/vsftpd.conf，命令如下：

```
sudo vi /etc/vsftpd.conf
```

打开以后 vsftpd.conf 文件以后找到如下两行：

```
local_enable=YES
write_enable=YES
```

确保上面两行前面没有“#”，有的话就取消掉，完成以后如图 4.1.1 所示：

```
27 # Uncomment this to allow local users to log in.
28 local_enable=YES
29 #
30 # Uncomment this to enable any form of FTP write command.
31 write_enable=YES
```

图 4.1.1 vsftpd.conf 修改

修改完 vsftpd.conf 以后保存退出，使用如下命令重启 FTP 服务：

```
sudo /etc/init.d/vsftpd restart
```

2、Windows 下 FTP 客户端安装

Windows 下 FTP 客户端我们使用 FileZilla，这是个免费的 FTP 客户端软件，可以在 FileZilla 官网下载，下载地址如下：https://www.filezilla.cn/download，下载界面如图 4.1.2 所示：



图 4.1.2 FileZilla 软件下载

我们已经下载好 FileZilla 并放到开发板光盘中了，路径为:3、软件->FileZilla_3.39.0_win64-setup_bundled.exe，双击安装即可。安装完成以后找到安装目录，找到图标，然后发送图标快捷方式到桌面，完成以后如图 4.1.3 所示：



图 4.1.3 FileZilla 图标

打开 FileZilla 软件，界面如图 4.1.4 所示：

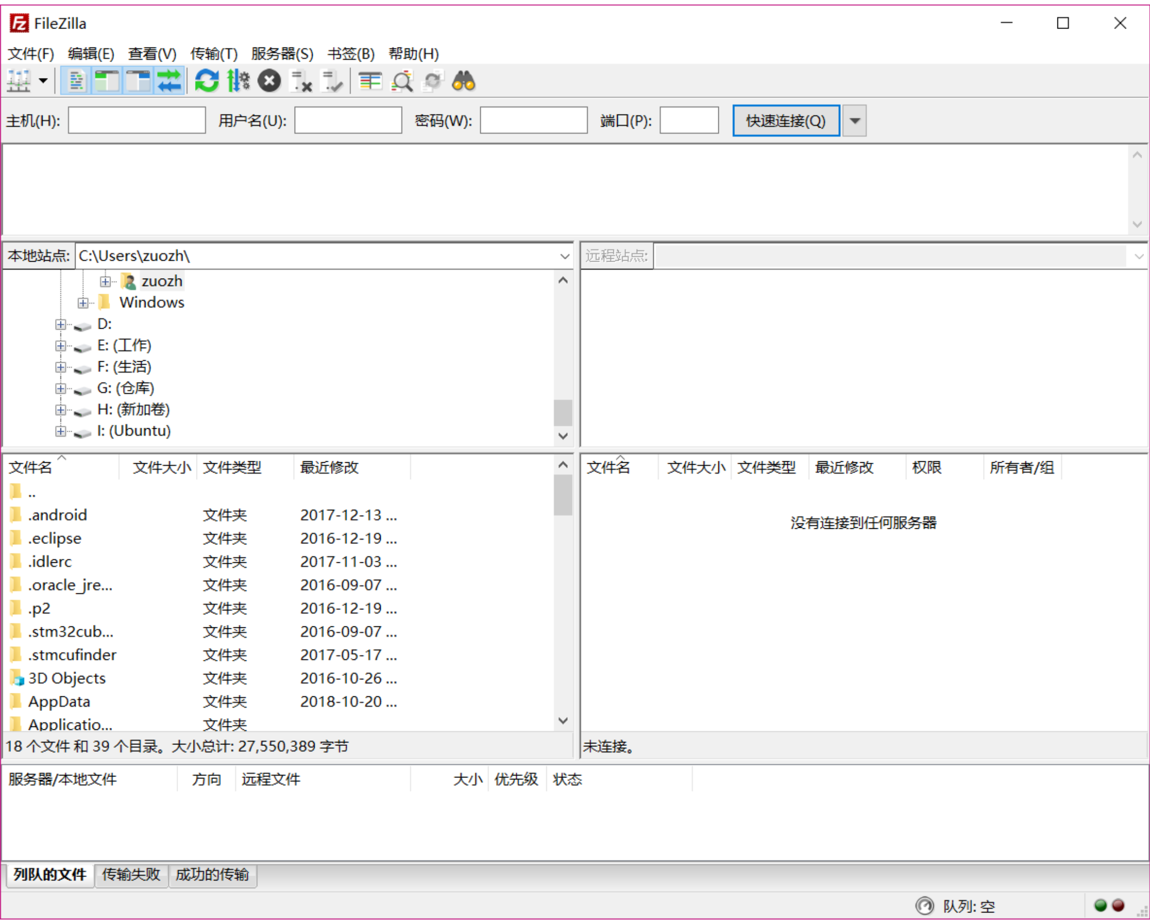


图 4.1.4 FileZilla 软件界面

3、FileZilla 软件设置

Ubuntu 作为 FTP 服务器，FileZilla 作为 FTP 客户端，客户端肯定要连接到服务器上，打开站点管理器，点击：文件->站点管理器，打开以后如图 4.1.5 所示：



图 4.1.5 站点管理器

点击图 4.1.5 中的“新站点(N)”按钮来创建站点，新建站点以后就会在“我的站点”下出现新建的这个站点，站点的名称可以自行修改，比如我将新的站点命名为“Ubuntu”如图 4.1.6 所示：

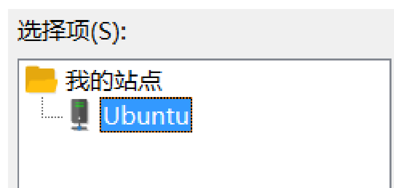


图 4.1.6 新建站点

选中新创建的“Ubuntu”站点，然后对站点的“常规”进行设置，设置如图 4.1.7 所示：



图 4.1.7 站点设置

按照图 4.1.7 中设置好以后，点击“连接”按钮，第一次连接可能会弹出提示是否保存密码的对话框，点击确定即可。连接成功以后如图 4.1.8 所示：

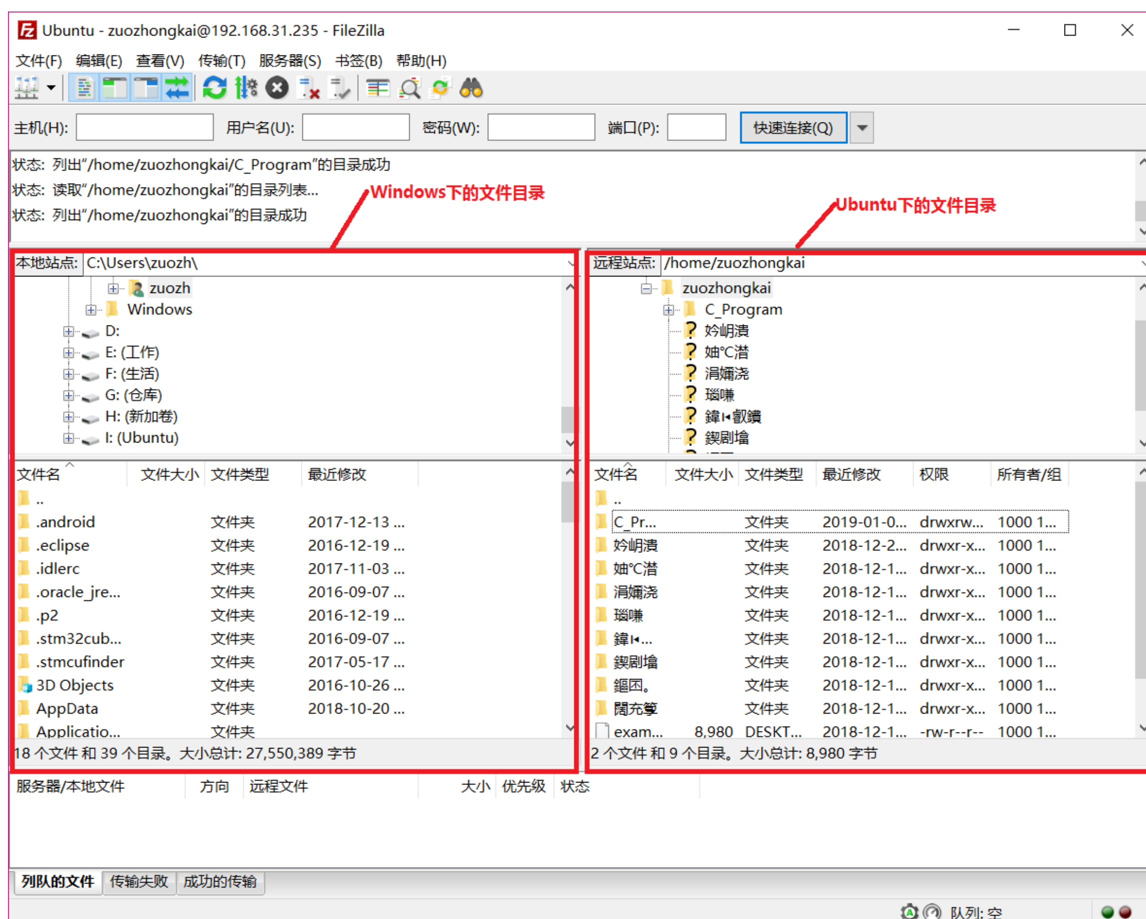


图 4.1.8 连接成功

连接成功以后如图 4.1.8 所示，其中左边就是 Windows 文件目录，右边是 Ubuntu 文件目录，默认进入用户根目录下（比如我电脑的“/home/zuozhongkai”）。但是注意观察在图 4.1.8 中 Ubuntu 文件目录下的中文目录都是乱码的，这是因为编码方式没有选对，先断开连接，点击：服务器(S)->断开连接，然后打开站点管理器，选中要设置的站点“Ubuntu”，选择“字符集”，设置如图 4.1.9 所示：

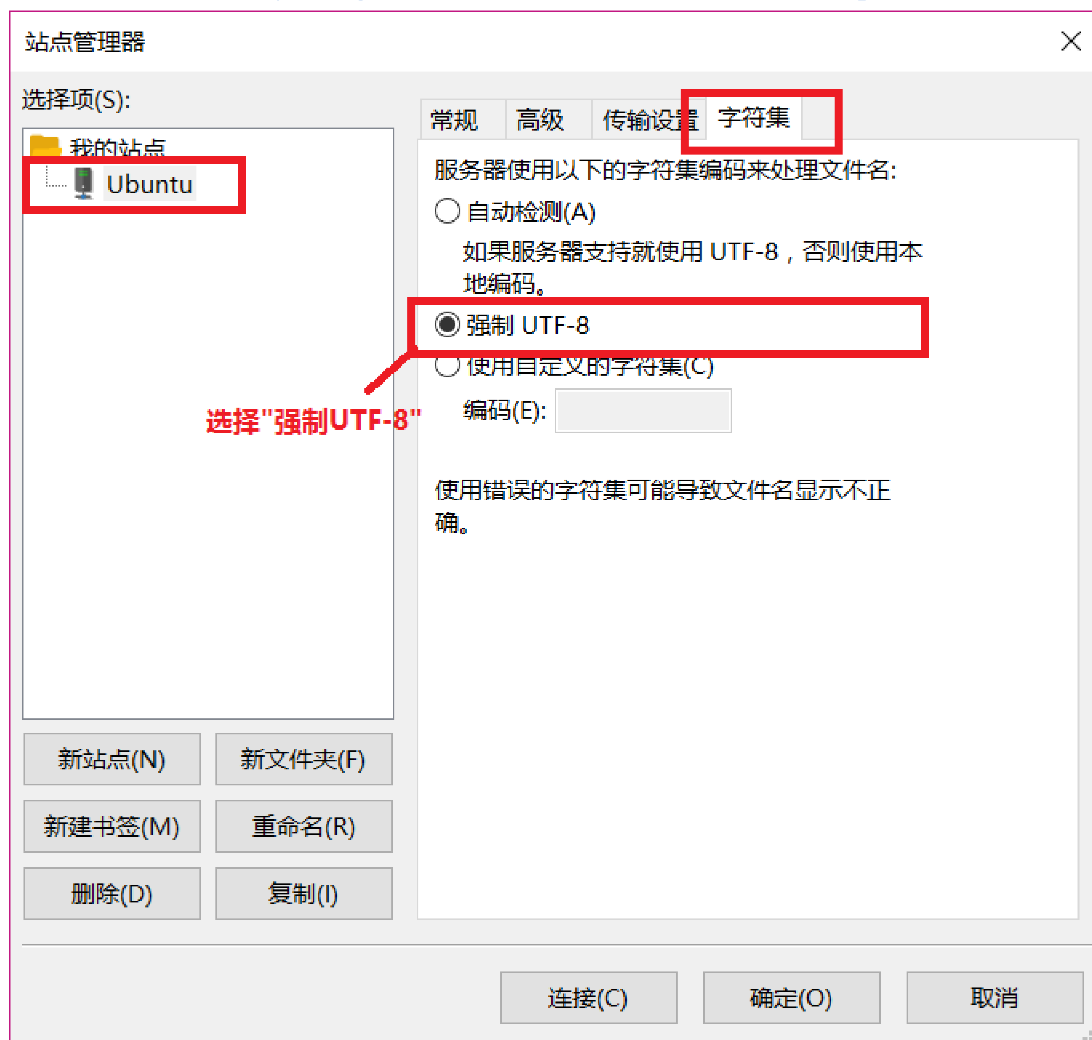


图 4.1.9 设置字符集

按照图 4.1.9 设置好字符集以后重新连接到 FTP 服务器上，重新链接到 FTP 服务器以后 Ubuntu 下的文件目录中文显示就正常了，如图 4.1.10 所示：

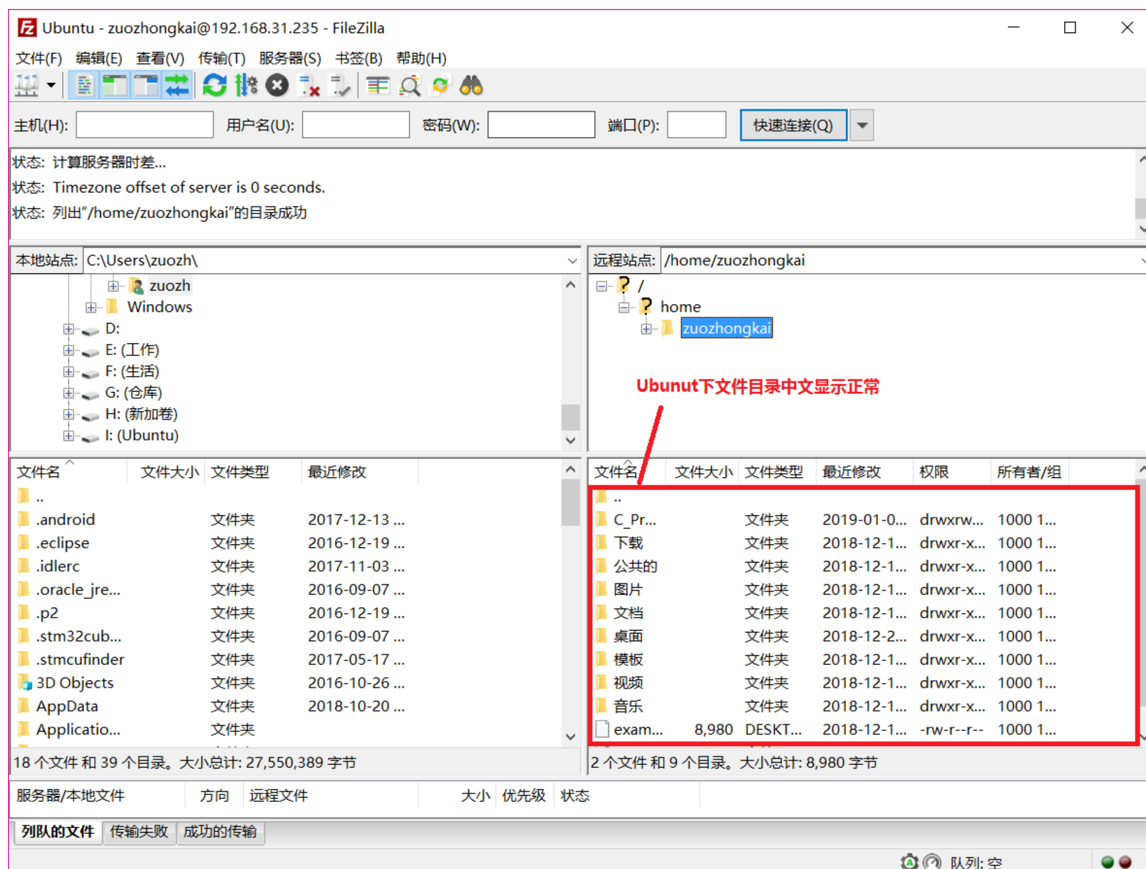


图 4.1.10 Ubuntu 下文件目录中文显示正常

如果要将 Windows 下的文件或文件夹拷贝到 Ubuntu 中，只需要在图 4.1.10 中左侧的 Windows 区域选中要拷贝的文件或者文件夹，然后直接拖到右侧的 Ubuntu 中指定的目录即可。将 Ubuntu 中的文件或者文件夹拷贝到 Windows 中也是直接拖放。

4.2 Ubuntu 下 NFS 和 SSH 服务开启

4.2.1 NFS 服务开启

后面进行 Linux 驱动开发的时候需要 NFS 启动，因此要先安装并开启 Ubuntu 中的 NFS 服务，使用如下命令安装 NFS 服务：

```
sudo apt-get install nfs-kernel-server portmap
```

等待安装完成，安装完成以后在用户根目录下创建一个名为“linux”的文件夹，以后所有的东西都放到这个“linux”文件夹里面，在“linux”文件夹里面新建一个名为“nfs”的文件夹，如图 4.2.1 所示：

```
zuozhongkai@ubuntu:~$ ls
C_Program  examples.desktop  linux  公共的  模板  视频  图片  文档  下载  音乐  桌面
zuozhongkai@ubuntu:~$ cd linux/
zuozhongkai@ubuntu:~/linux$ ls
nfs
zuozhongkai@ubuntu:~/linux$
```

图 4.2.1 创建 linux 工作目录

图 4.2.1 中创建的 nfs 文件夹供 nfs 服务器使用，以后我们可以在开发板上通过网络文件系统来访问 nfs 文件夹，要先配置 nfs，使用如下命令打开 nfs 配置文件/etc/exports：

```
sudo vi /etc/exports
```

打开/etc/exports 以后在后面添加如下所示内容：

```
/home/zuozhongkai/linux/nfs *(rw,sync,no_root_squash)
```

添加完成以后的/etc/exports 如图 4.2.2 所示：

```
 1 # /etc/exports: the access control list for filesystems which may be exported
 2 #               to NFS clients.  See exports(5).
 3 #
 4 # Example for NFSv2 and NFSv3:
 5 # /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
 6 #
 7 # Example for NFSv4:
 8 # /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
 9 # /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
10 #
11
12 /home/zuozhongkai/linux/nfs *(rw,sync,no_root_squash)
```

图 4.2.2 修改文件/etc/exports

重启 NFS 服务，使用命令如下：

```
sudo /etc/init.d/nfs-kernel-server restart
```

4.2.2 SSH 服务开启

开启 Ubuntu 的 SSH 服务以后我们就可以在 Windwos 下使用终端软件登陆到 Ubuntu，比如使用 SecureCRT，Ubuntu 下使用如下命令开启 SSH 服务：

```
sudo apt-get install openssh-server
```

上述命令安装 ssh 服务，ssh 的配置文件为/etc/ssh/sshd_config，使用默认配置即可。

4.3 Ubuntu 交叉编译工具链安装

4.3.1 交叉编译器安装

ARM 裸机、Uboot 移植、Linux 移植这些都需要在 Ubuntu 下进行编译，编译就需要编译器，我们在第三章“Linux C 编程入门”里面已经讲解了如何在 Liux 进行 C 语言开发，里面使用 GCC 编译器进行代码编译，但是 Ubuntu 自带的 gcc 编译器是针对 X86 架构的！而我们现在要编译的是 ARM 架构的代码，所以我们需要一个在 X86 架构的 PC 上运行，可以编译 ARM 架构代码的 GCC 编译器，这个编译器就叫做交叉编译器，总结一下交叉编译器就是：

1、它肯定是一个 GCC 编译器。

2、这个 GCC 编译器是运行在 X86 架构的 PC 上的。

3、这个 GCC 编译器是编译 ARM 架构代码的，也就是编译出来的可执行文件是在 ARM 芯片上运行的。

交叉编译器中“交叉”的意思就是在一个架构上编译另外一个架构的代码，相当于两种架构“交叉”起来了。

交叉编译器有很多种，我们使用 Linaro 出品的交叉编译器，Linaro 一间非营利性质的开放源代码软件工程公司，Linaro 开发了很多软件，最著名的就是 Linaro GCC 编译工具链(编译器)，关于 Linaro 详细的介绍可以到 Linaro 官网查阅。Linaro GCC 编译器下载地址如下：
https://releases.linaro.org/components/toolchain/binaries/latest-7/arm-linux-gnueabihf/，打开以后下载界面如图 4.3.1.1 所示：

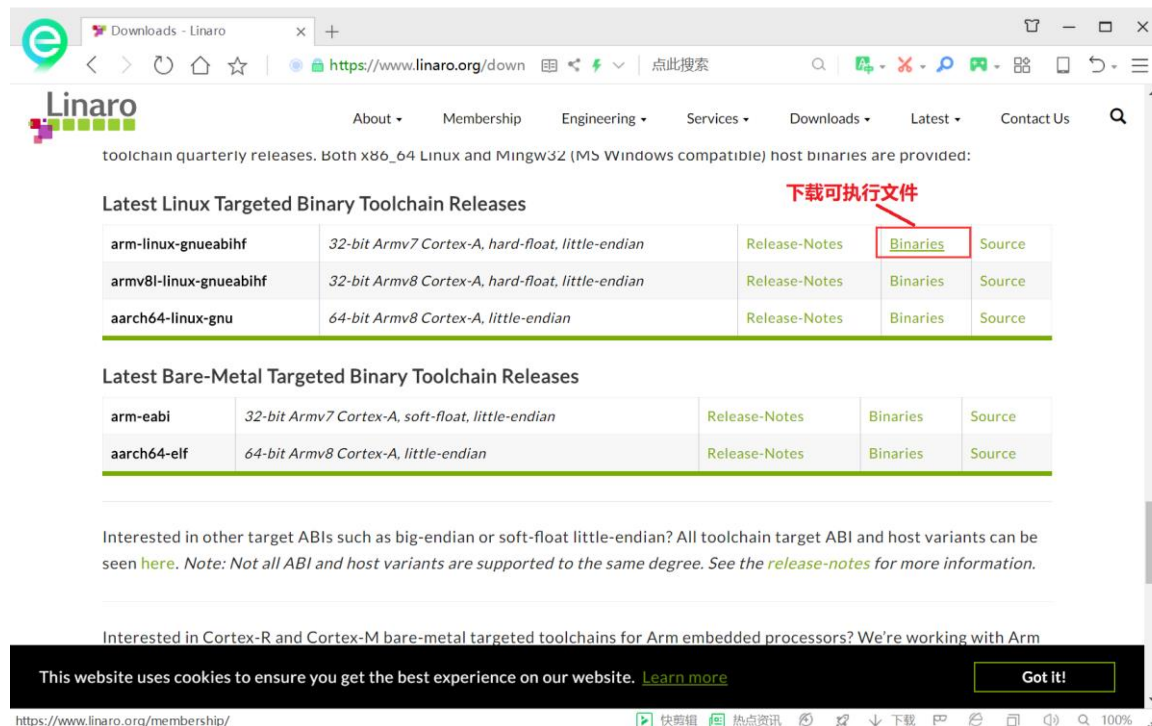


图 4.3.1.1 Linaro 下载界面

在图 4.3.1.1 中有很多种 GCC 交叉编译工具链，因为我们所使用的 I.MX6U-ALPHA 开发板是一个 Cortex-A7 内核的开发板，因此选择 arm-linux-gnueabihf，点击后面的“Binaries”进入可执行文件下载界面，如图 4.3.1.2 所示：

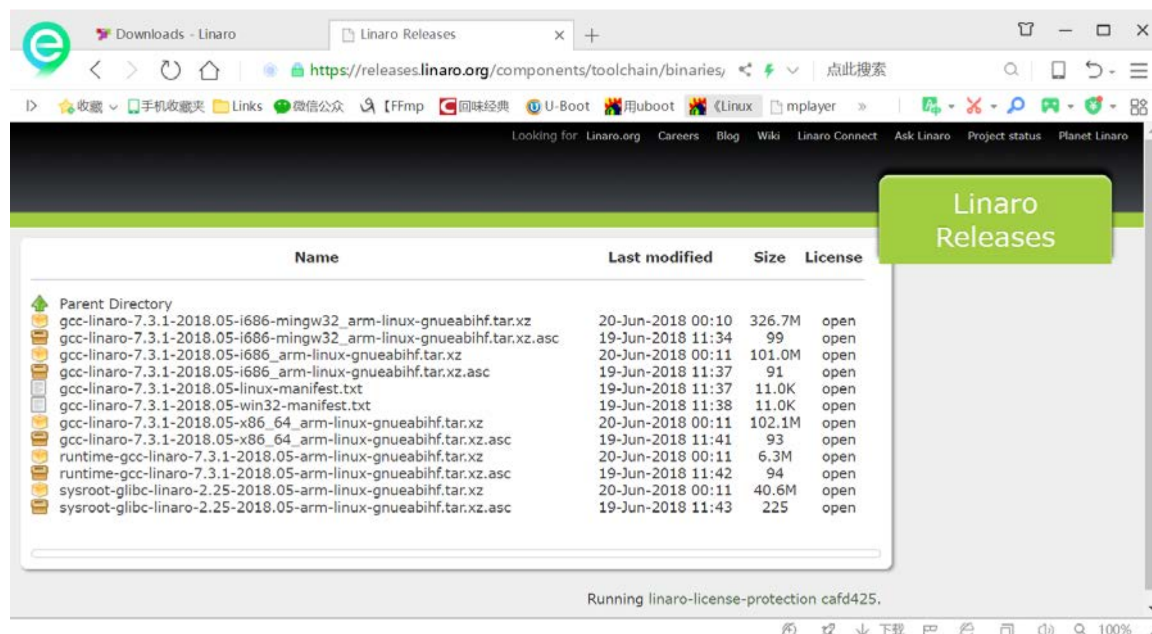


图 4.3.1.2 Linaro 交叉编译器下载

在写本教程的时候最新的编译器版本是 7.3.1，但是笔者在测试 7.3.1 版本编译器的时候发现编译完成后的 uboot 无法运行。所以这里不推荐使用最新版的编译器。笔者测试过 4.9 版本的编译器可以正常工作，所以我们需要下载 4.9 版本的编译器，下载地址为：https://releases.linaro.org/components/toolchain/binaries/4.9-2017.01/arm-linux-gnueabihf/ ，如图

4.3.1.3 所示：

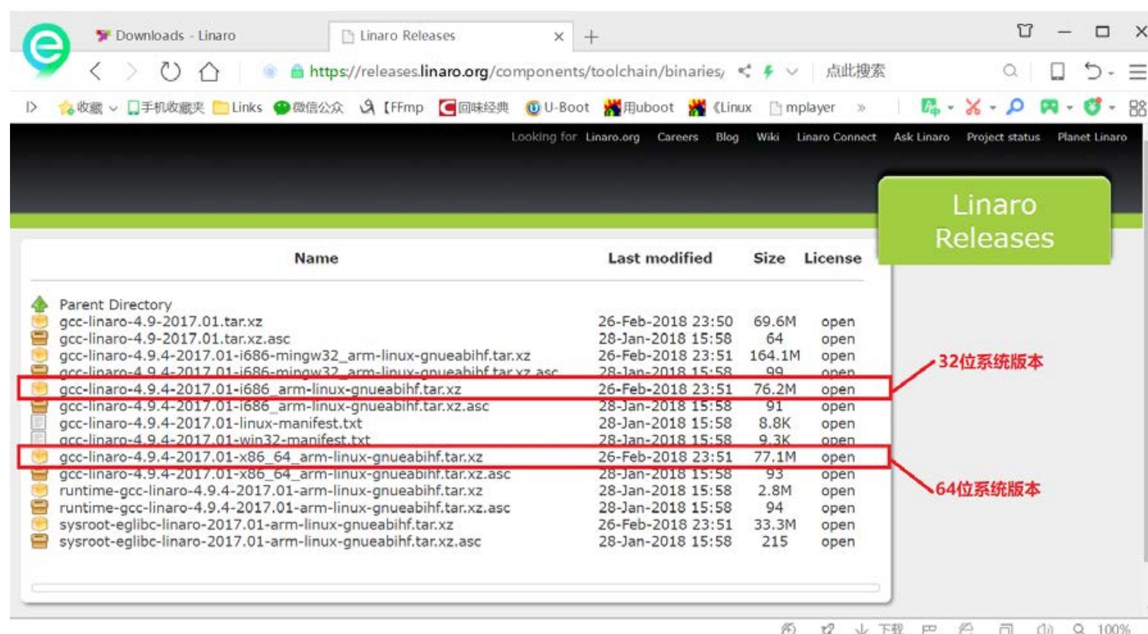


图 4.3.1.3 4.9.4 版本编译器下载

图 4.3.1.3 中有很多种交叉编译器，我们只需要关注这两种：gcc-linaro-4.9.4-2017.01-i686_arm-linux-gnueabihf.tar.tar.xz 和 gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz，第一个是针对 32 位系统的，第二个是针对 64 位系统的。大家根据自己所使用的 Ubuntu 系统类型选择合适的版本，比如我安装的 Ubuntu 16.04 是 64 位系统，因此我要使用 gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz。

这两种交叉编译器我们已经下载好放到了开发板光盘中，路径：5、开发工具->1、交叉编译器。我们要先将交叉编译工具拷贝到 Ubuntu 中，在 4.2.1 小节中我们在当前用户根目录下创建了一个名为“linux”的文件夹，在这个 linux 文件夹里面再创建一个名为“tool”的文件夹，用来存放一些开发工具。使用前面已经安装好的 FileZilla 将交叉编译器拷贝到 Ubuntu 中刚刚新建的“tool”文件夹中，操作如图 4.3.1.4 所示：

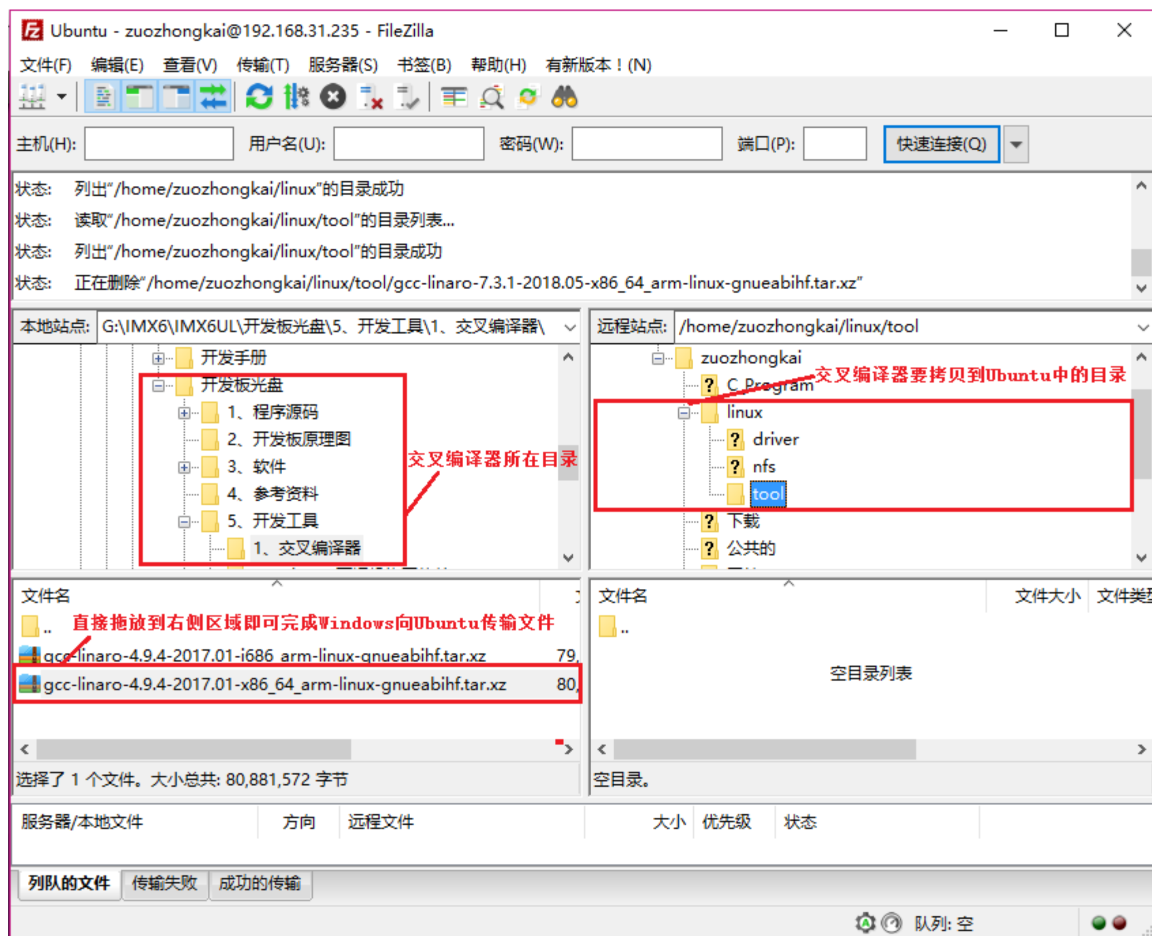


图 4.3.1.4 拷贝交叉编译器

拷贝完成的话 FileZilla 会有提示，如图 4.3.1.5：

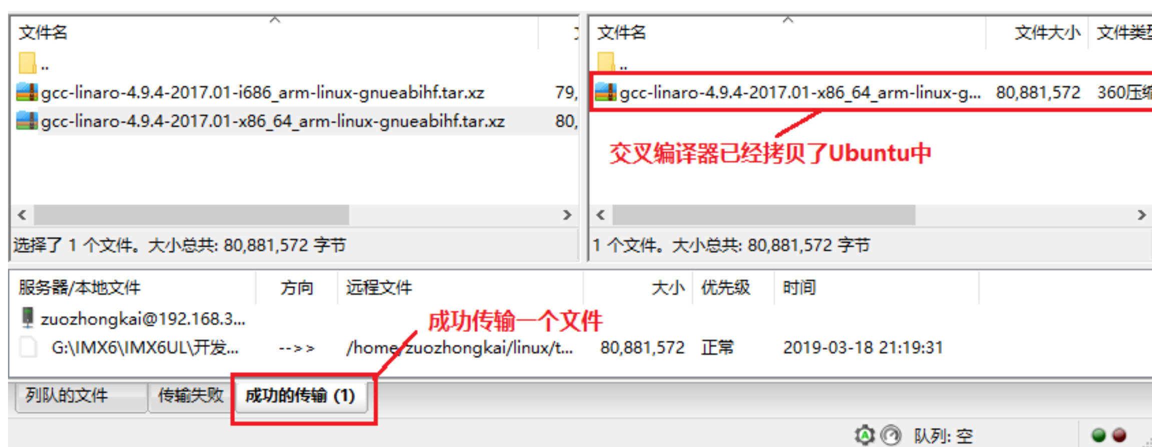


图 4.3.1.5 交叉编译器拷贝完成

在 Ubuntu 中创建目录：/usr/local/arm，命令如下：

```
sudo mkdir /usr/local/arm
```

创建完成以后将刚刚拷贝的交叉编译器复制到/usr/local/arm 这个目录中，在终端使用命令“cd”进入到存放有交叉编译器的目录，比我前面将交叉编译器拷贝到了目录“/home/zuozhongkai/linux/tool”中，然后使用如下命令将交叉编译器复制到/usr/local/arm 中：

```
sudo cp gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz /usr/local/arm/ -f
```

操作步骤如图 4.3.1.6 所示：

```
zuozhongkai@ubuntu:~/linux/tool$ ls                    //查看交叉编译工具链是否存在
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz
zuozhongkai@ubuntu:~/linux/tool$ sudo cp gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz /usr/local/arm/ -f

zuozhongkai@ubuntu:~/linux/tool$
zuozhongkai@ubuntu:~/linux/tool$ cd /usr/local/arm/    //进入/usr/local/arm文件夹
zuozhongkai@ubuntu:/usr/local/arm$ ls                  //查看交叉编译器工具链是否拷贝到了/usr/local/arm文件夹中
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz
zuozhongkai@ubuntu:/usr/local/arm$
```

图 4.3.1.6 拷贝交叉编译工具到/usr/local/arm 目录中

拷贝完成以后在/usr/local/arm 目录中对交叉编译工具进行解压，解压命令如下：

```
sudo tar -vxf gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf.tar.xz
```

等待解压完成，解压完成以后会生成一个名为“gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf”的文件夹，这个文件夹里面就是我们的交叉编译工具链。

修改环境变量，使用 VI 打开/etc/profile 文件，命令如下：

```
sudo vi /etc/profile
```

打开/etc/profile 以后，在最后面输入如下所示内容：

```
export PATH=$PATH:/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin
```

添加完成以后的/etc/profile 如图 4.3.1.7 所示：

```
zuozhongkai@ubuntu: ~
  8     if [ -f /etc/bash.bashrc ]; then
  9       . /etc/bash.bashrc
 10     fi
 11   else
 12     if [ "`id -u`" -eq 0 ]; then
 13       PS1='# '
 14     else
 15       PS1='$ '
 16     fi
 17   fi
 18 fi
 19
 20 if [ -d /etc/profile.d ]; then
 21   for i in /etc/profile.d/*.sh; do
 22     if [ -r $i ]; then
 23       . $i
 24     fi
 25   done
 26   unset i
 27 fi
 28
 29 export PATH=$PATH:/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin
 30
-- 插入 --                                                        30,1          底端
```

图 4.3.1.7 添加环境变量

修改好以后就保存退出，重启 Ubuntu 系统，交叉编译工具链(编译器)就安装成功了。

4.3.2 安装相关库

在使用交叉编译器之前还需要安装一下其它的库，命令如下：

```
sudo apt-get install lsb-core lib32stdc++6
```

等待这些库安装完成。

4.3.3 交叉编译器验证

首先查看一下交叉编译工具的版本号，输入如下命令：

```
arm-linux-gnueabihf-gcc －v
```

如果交叉编译器安装正确的话就会显示版本号，如图 4.3.3.1 所示：

```
zuozhongkai@ubuntu: ~
zuozhongkai@ubuntu:~$ arm-linux-gnueabihf-gcc -v
Using built-in specs.
COLLECT_GCC=arm-linux-gnueabihf-gcc
COLLECT_LTO_WRAPPER=/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../libexec/gcc/arm-linux-gnueabihf/4.9
.4/lto-wrapper
Target: arm-linux-gnueabihf
Configured with: /home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabih
f/snapshots/gcc-linaro-4.9-2017.01/configure SHELL=/bin/bash --with-mpc=/home/tcwg-buildslave/workspace/tcwg-make-release/label/d
ocker-trusty-amd64-tcwg-build/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu --with-mpfr=/home/tcwg-bu
ildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabihf/_build/builds/destdir/x86_64
-unknown-linux-gnu --with-gmp=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-l
inux-gnueabihf/_build/builds/destdir/x86_64-unknown-linux-gnu --with-gnu-as --with-gnu-ld --disable-libmudflap --enable-lto --ena
ble-objc-gc --enable-shared --without-included-gettext --enable-nls --disable-sjlj-exceptions --enable-gnu-unique-object --enable
-linker-build-id --disable-libstdcxx-pch --enable-c99 --enable-clocale=gnu --enable-libstdcxx-debug --enable-long-long --with-clo
og=no --with-ppl=no --with-isl=no --disable-multilib --with-float=hard --with-mode=thumb --with-tune=cortex-a9 --with-arch=armv7-
a --with-fpu=vfpv3-d16 --enable-threads=posix --enable-multiarch --enable-libstdcxx-time=yes --with-build-sysroot=/home/tcwg-buil
dslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabihf/_build/sysroots/arm-linux-gnue
abihf --with-sysroot=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnue
abihf/_build/builds/destdir/x86_64-unknown-linux-gnu/arm-linux-gnueabihf/libc --enable-checking=release --disable-bootstrap --ena
ble-languages=c,c++,fortran,lto --build=x86_64-unknown-linux-gnu --host=x86_64-unknown-linux-gnu --target=arm-linux-gnueabihf --p
refix=/home/tcwg-buildslave/workspace/tcwg-make-release/label/docker-trusty-amd64-tcwg-build/target/arm-linux-gnueabihf/_build/bu
ilds/destdir/x86_64-unknown-linux-gnu
Thread model: posix
gcc version 4.9.4 (Linaro GCC 4.9-2017.01)
zuozhongkai@ubuntu:~$
```

图 4.3.3.1 交叉编译器版本查询

从图 4.3.3.1 中可以看出当前交叉编译器的版本号为 4.9.4，说明交叉编译工具链安装成功。第三章“Linux C 编程入门”中使用 Ubuntu 自带的 GCC 编译器，我们用的是命令“gcc”。要使用刚刚安装的交叉编译器的时候使用的命令是“arm-linux-gnueabihf-gcc”，“arm-linux-gnueabihf-gcc”的含义如下：

1、arm 表示这是编译 arm 架构代码的编译器。

2、linux 表示运行在 linux 环境下。

3、gnueabihf 表示嵌入式二进制接口。

4、gcc 表示是 gcc 工具。

最好的验证验证方法就是直接编译一个例程，我们就编译第一个裸机例程“1_leds”试试，裸机例程在开发板光盘中的路径为：1、程序源码->1、裸机例程-> 1_leds。在前面创建的 linux 文件夹下创建 driver/board_driver 文件夹，用来存放裸机例程，如图 4.3.3.2 所示：

```
zuozhongkai@ubuntu:~/linux/driver$ ls
board_driver
```

图 4.3.3.2 创建 board_driver 文件夹

将第一个裸机例程“1_leds”拷贝到 board_driver 中，然后执行 make 命令进行编译，如图 4.3.3.3 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls     //检查Makefile是否存在
imxdownload  led.bin  led.dis  led.elf  led.o  led.s  load.imx  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ make   //使用make命令编译工程
arm-linux-gnueabihf-gcc -g -c -o led.o led.s
arm-linux-gnueabihf-ld -Ttext 0X87800000 -g led.o -o led.elf
arm-linux-gnueabihf-objcopy -O binary -S led.elf led.bin
arm-linux-gnueabihf-objdump -D led.elf > led.dis
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls     //检查编译结果
imxdownload  led.bin  led.dis  led.elf  led.o  led.s  load.imx  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 4.3.3.3 编译过程

从图 4.3.3.3 可以看到例程“1_leds”编译成功了，编译生成了 led.o 和 led.bin 这两个文件，使用如下命令查看 led.o 文件信息：

```
file led.o
```

结果如图 4.3.3.4 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ file led.o
led.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1 (SYSV), not stripped
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 4.3.3.4 led.o 文件信息

从图 4.3.3.4 可以看到 led.o 是 32 位 LSB 的 ELF 格式文件，目标机架构为 ARM，说明我们的交叉编译器工作正常。

4.4 Source Insight 软件安装和使用

4.4.1 Source Insight 安装

Source Inisght 是一款功能强大的代码编辑、阅读工具，工作在 Windows 下，我们可以用 Source Insight 来进行代码编写和阅读，编写完成以后将代码拷贝到 Ubuntu 中去编译即可。Source Insight 下载地址为：https://www.sourceinsight.com/download/，如图 4.4.1.1 所示：

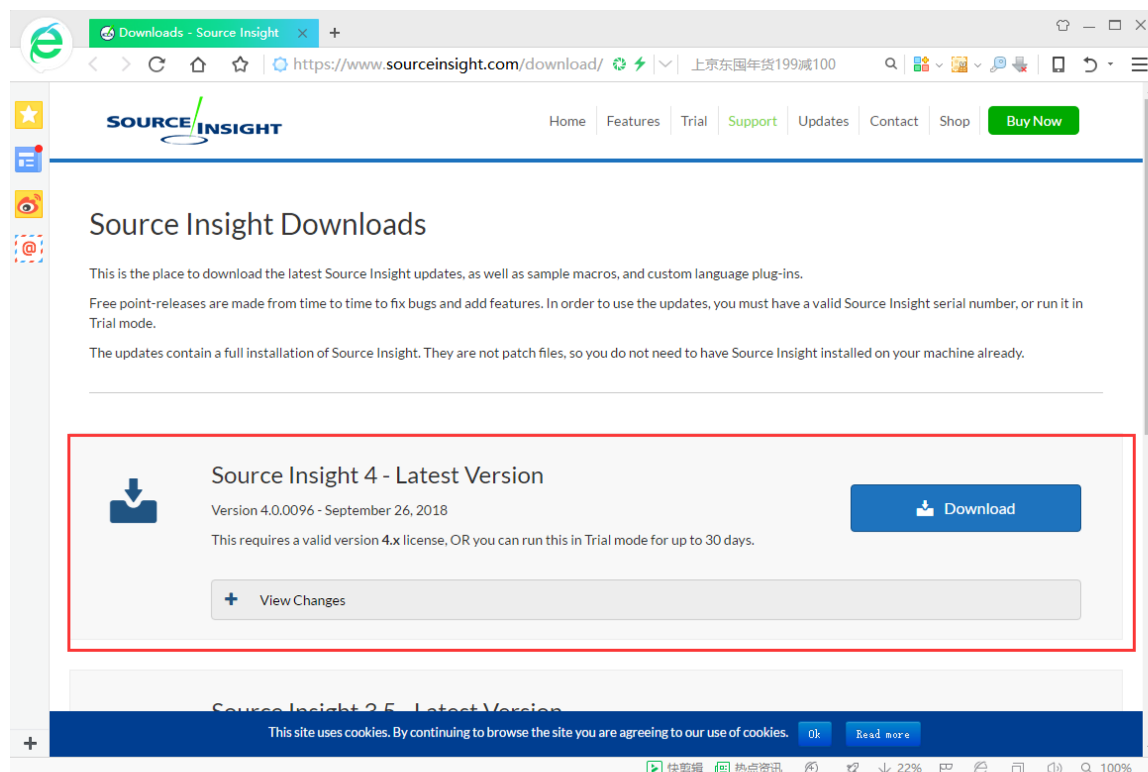


图 4.4.1.1 Source Insight 下载界面

我们已经下载好并放到了开发板光盘中，路径为：3、软件->Source Insight 4.0->sourceinsight4096-setup.exe，双击“sourceinsight4096-setup.exe”即可开始安装，首先是图 4.4.1.2 所示欢迎界面：

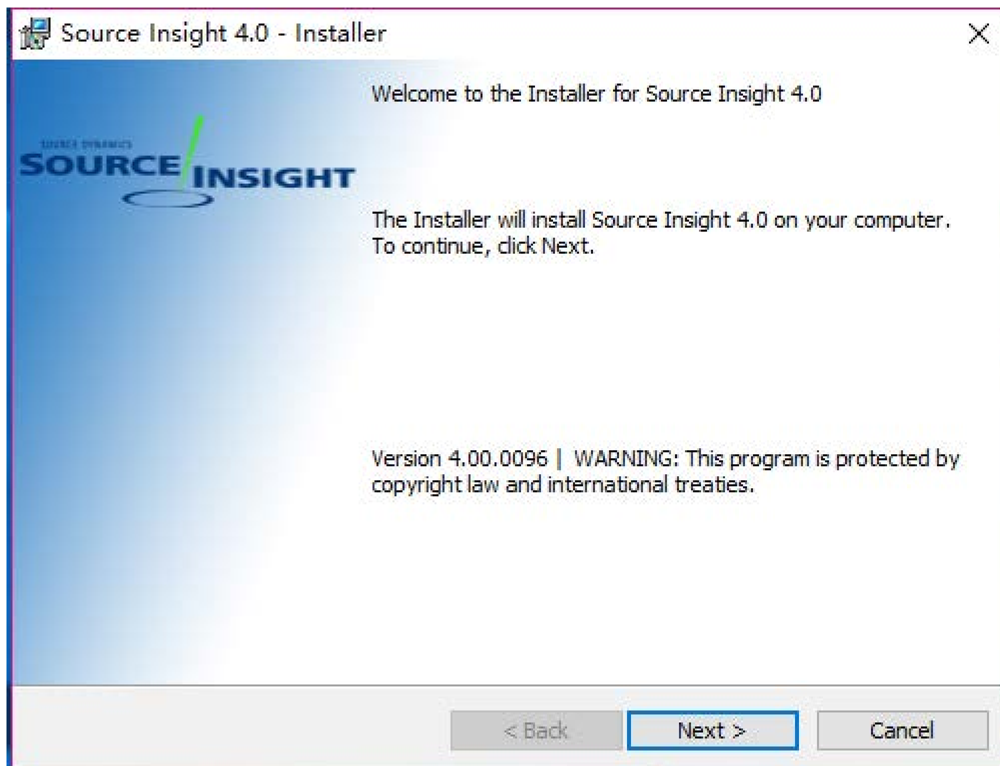


图 4.4.1.2 Souce Insight 4.0 安装欢迎界面

点击图 4.4.1.2 中的“Next”按钮进入下一步，如图 4.4.1.3 所示：

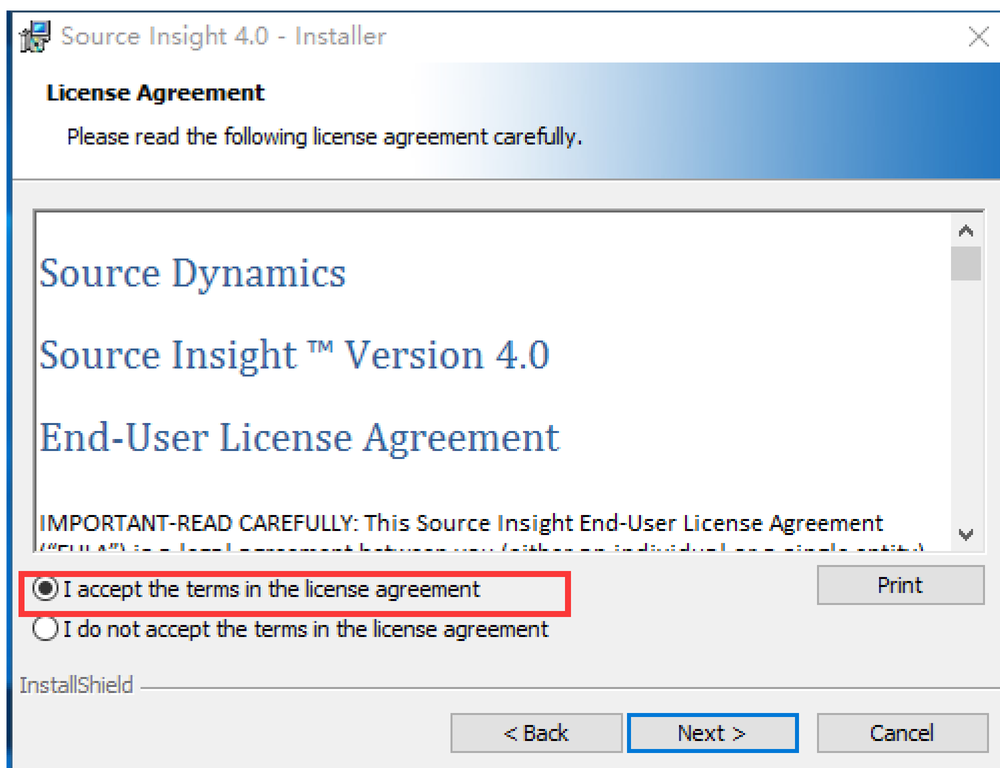


图 4.4.1.3 条例许可界面

选择图 4.4.1.3 中的“I accept the terms in the license adreement”，然后点击“Next”按钮，进入安装目录选择界面，根据自己的实际情况选择合适的安装目录，如图 4.4.1.4 所示：

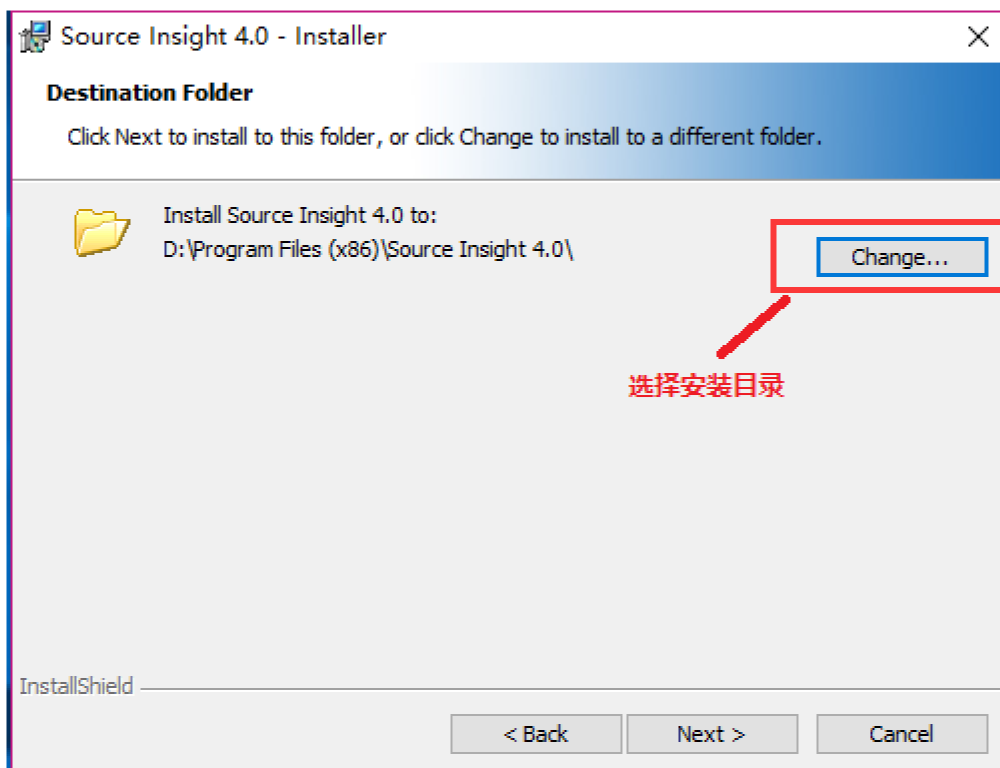


图 4.4.1.4 安装目录选择

选择好安装目录以后点击“Next”按钮，进入图 4.4.1.5 所示的准备安装界面：

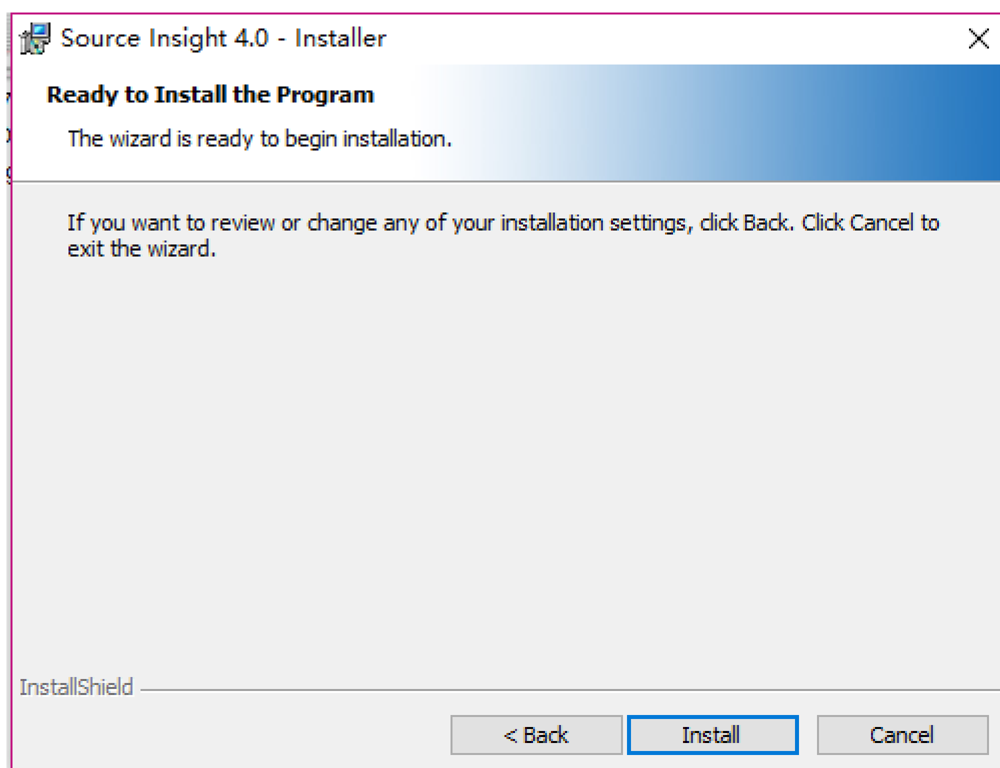


图 4.4.1.5 准备安装界面

点击图 4.4.1.5 中的“Install”按钮开始安装，等待安装完成，安装完成以后如图 4.4.1.6 所示：

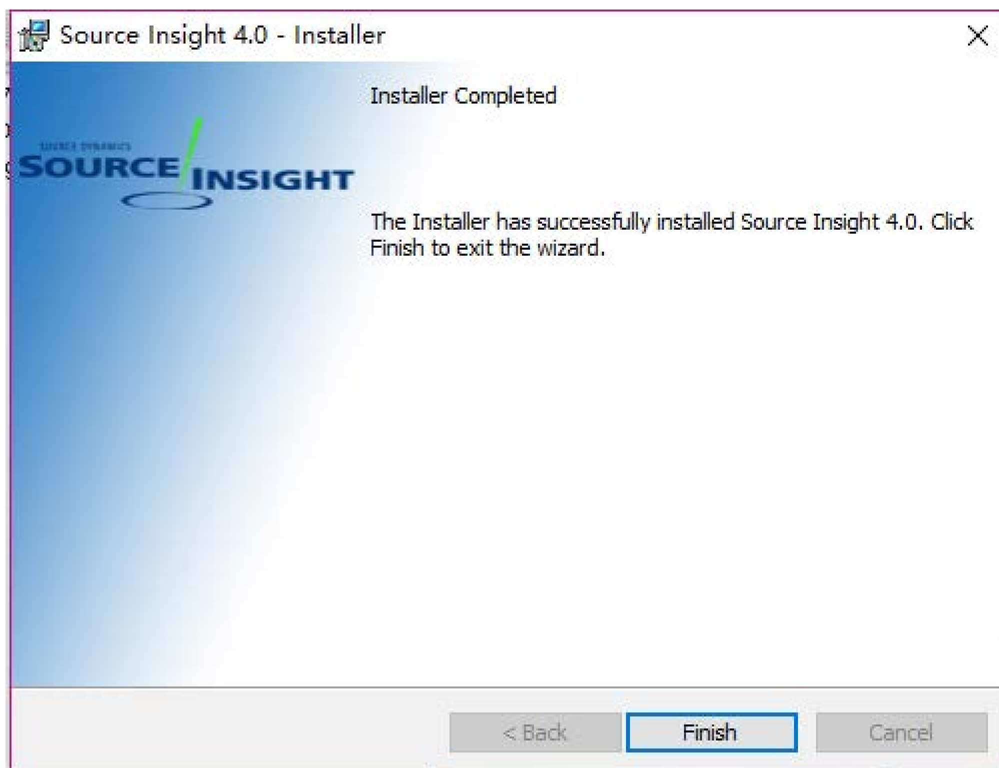


图 4.4.1.6 安装完成界面

点击图 4.4.1.6 中的“Finish”按钮退出安装，安装成功以后会在桌面上出现 Source Insight 4.0 的图标，如图 4.4.1.7 所示：



图 4.4.1.7 Sourc Insight 4.0 图标

双击图标打开 Source Inisght 4.0，第一次打开的话会有 Licese 提示，如图 4.4.1.8 所示：

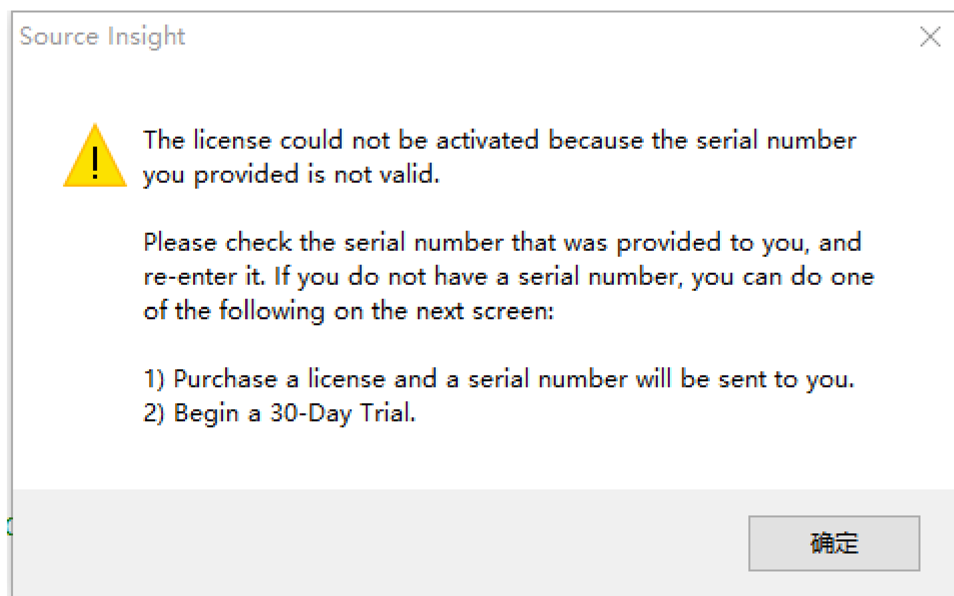


图 4.4.1.9 Licese 提示

因为 Source Insight 4.0 是个收费软件，所以是需要购买 License 的，如果没有购买的话可以免费体验 30 天，点击图 4.4.1.9 中的“确定”按钮，进入图 4.4.1.10 所示界面：

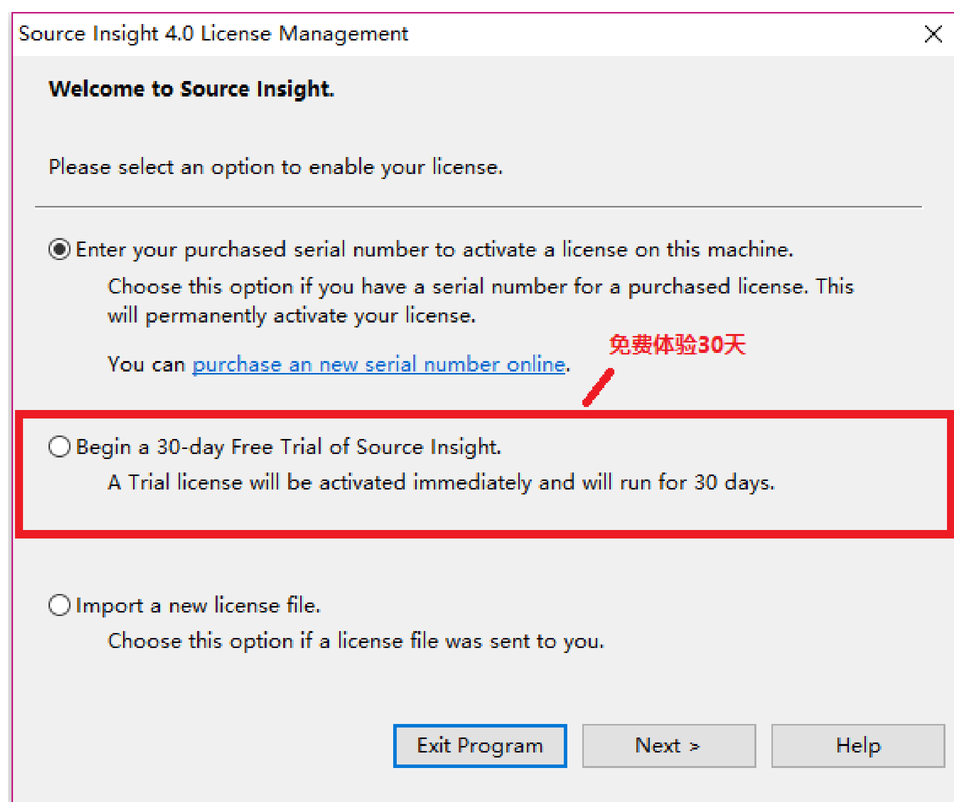


图 4.4.1.10 license 输入界面

在图 4.4.1.10 中，如果你已经购买了 licese 那么就选择第一个，如果没有购买 licese 的话就选择第二个免费体验 30 天，选择好以后点击“Next”按钮，进入图 4.4.1.11 所示界面：

Source Insight Trial Activation

Please enter your information below. When you continue, a Trial license will be activated on your machine. This requires Internet access.

Your Name: *

Company or Organization:

Your email: *

Re-type email:

< Back　Next >

图 4.4.1.11 信息输入界面

填写图 4.4.1.11 中的信息，然后点击“Next”，填写好以后一路“Next”下去就可以了，打开以后的默认界面如图 4.4.1.12 所示：

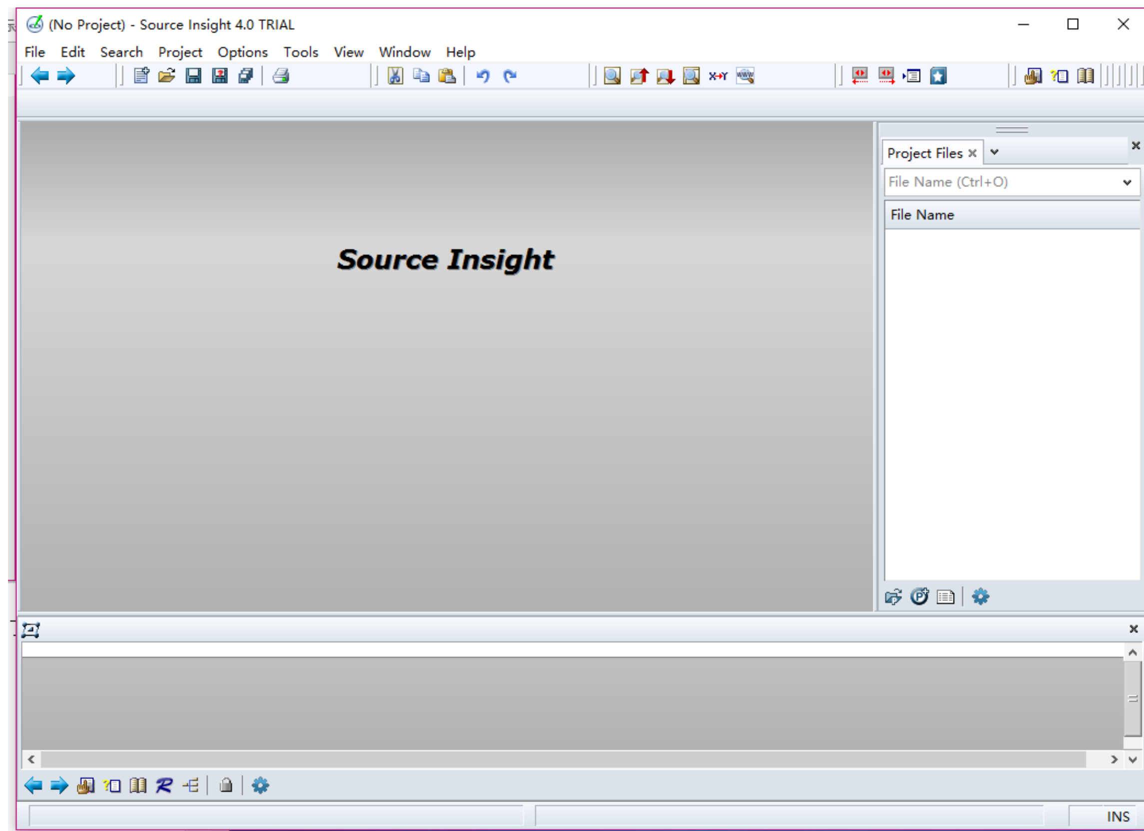


图 4.4.1.12 Source Insight 默认界面

至此 Source Insight 安装完成。

4.4.2 Source Insight 新建工程

1、新建工程

跟 MDK、IAR 一样，Source Insight 是需要创建工程的，但是远没有 MDK 和 IAR 那么复杂，先新建一个工程文件夹， 比如 test，test 用来存放工程所有文件，包括 Source Insight 工程文件和 C 语言源码文件。

注意！Source Insight 的工程不能有中文路径!!!!!!

注意！Source Insight 的工程不能有中文路径!!!!!!

注意！Source Insight 的工程不能有中文路径!!!!!!

在刚刚创建的 test 文件夹中新建一个 SI 文件夹，用来存放 Source Insight 的所有工程文件，完成以后如图 4.4.2.1 所示：

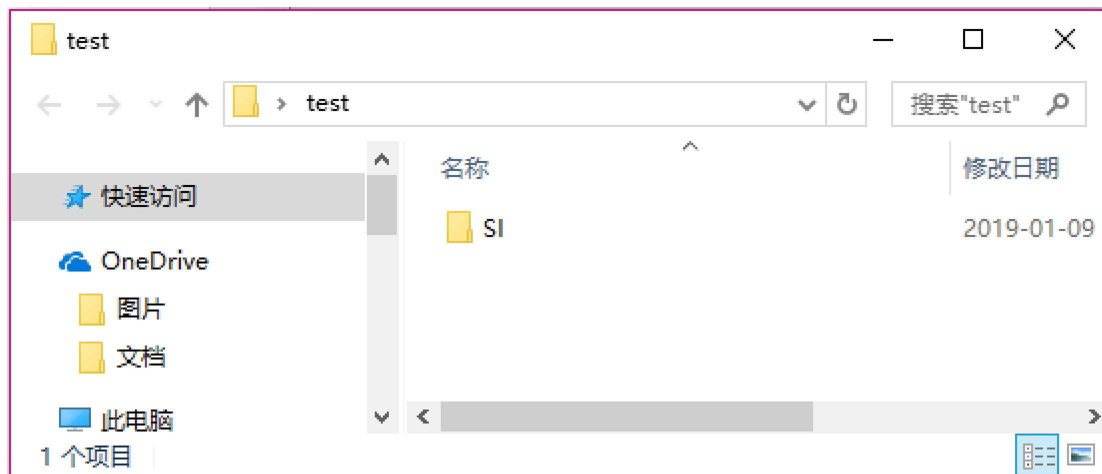


图 4.4.2.1 工程文件目录

工程文件夹准备好以后就可以创建工程了，点击 Source Insight 的：Project->New Project，如图 4.4.2.2 所示：

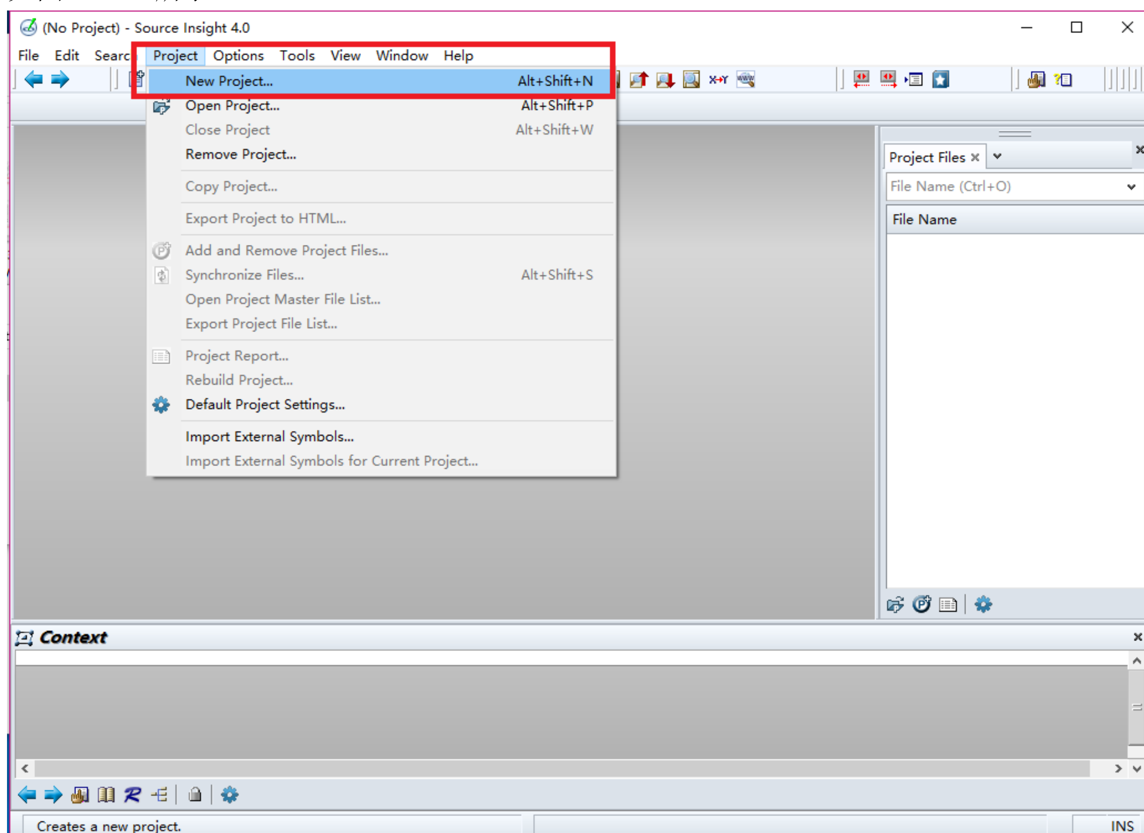


图 4.4.2.2 新建工程

点击“New Project”后进入图 4.4.2.3 所示界面：



图 4.4.2.3 工程名字和路径设置

在图 4.4.2.3 中设置好工程名字和路径以后点击“OK”按钮，会进入另外一个设置界面，如图 4.4.2.4 所示：

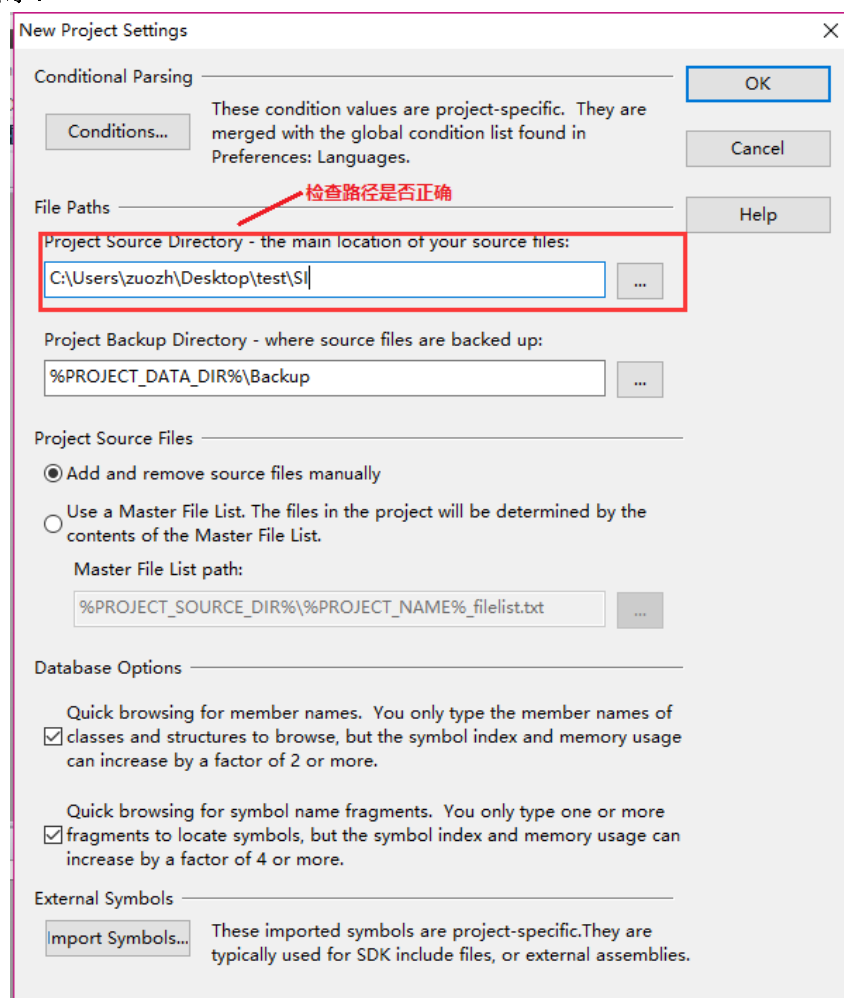


图 4.4.2.4 工程设置 2

在图 4.4.2.4 中我们一般不需要做任何修改，主要是检查一下路径是否正确，如果没问题的话就点击“OK”按钮即可，进入向工程添加文件界面，如图 4.4.2.5 所示：

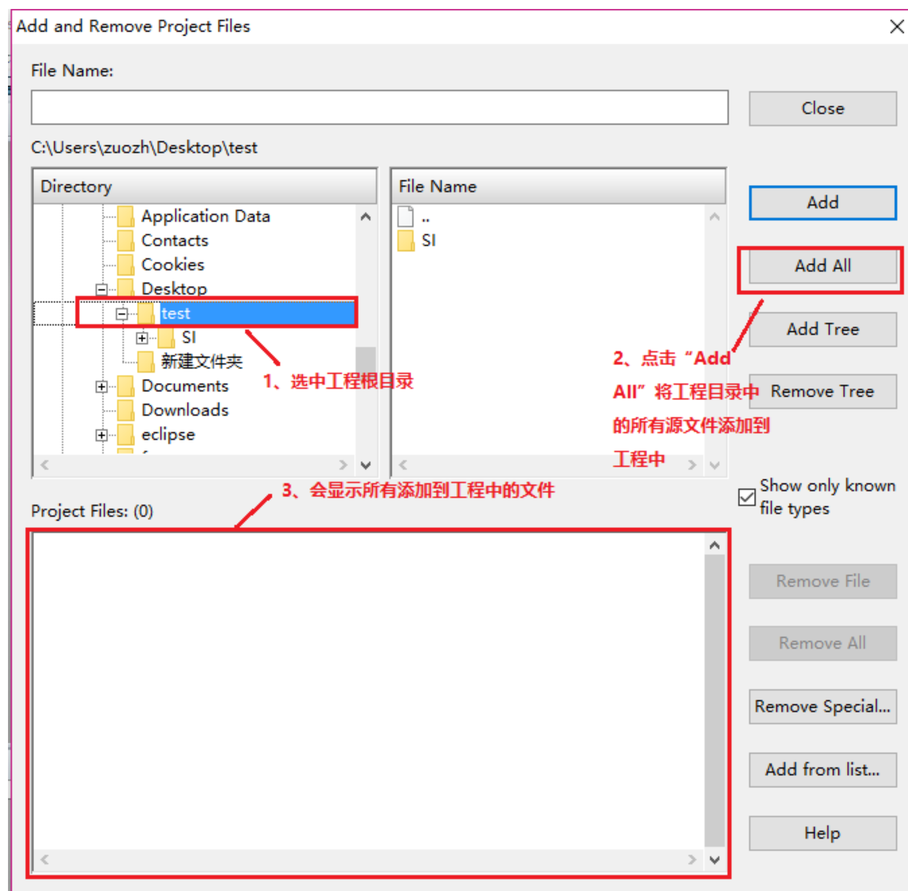


图 4.4.2.5 想工程添加源文件

如果你的工程文件夹已经有源文件了，那么就可以按照图 4.4.2.5 所示方法将所有的源文件添加到工程中，添加完成以后点击“Close”按钮关闭即可。新建工程完成以后 Source Insight 如图 4.4.2.6 所示：

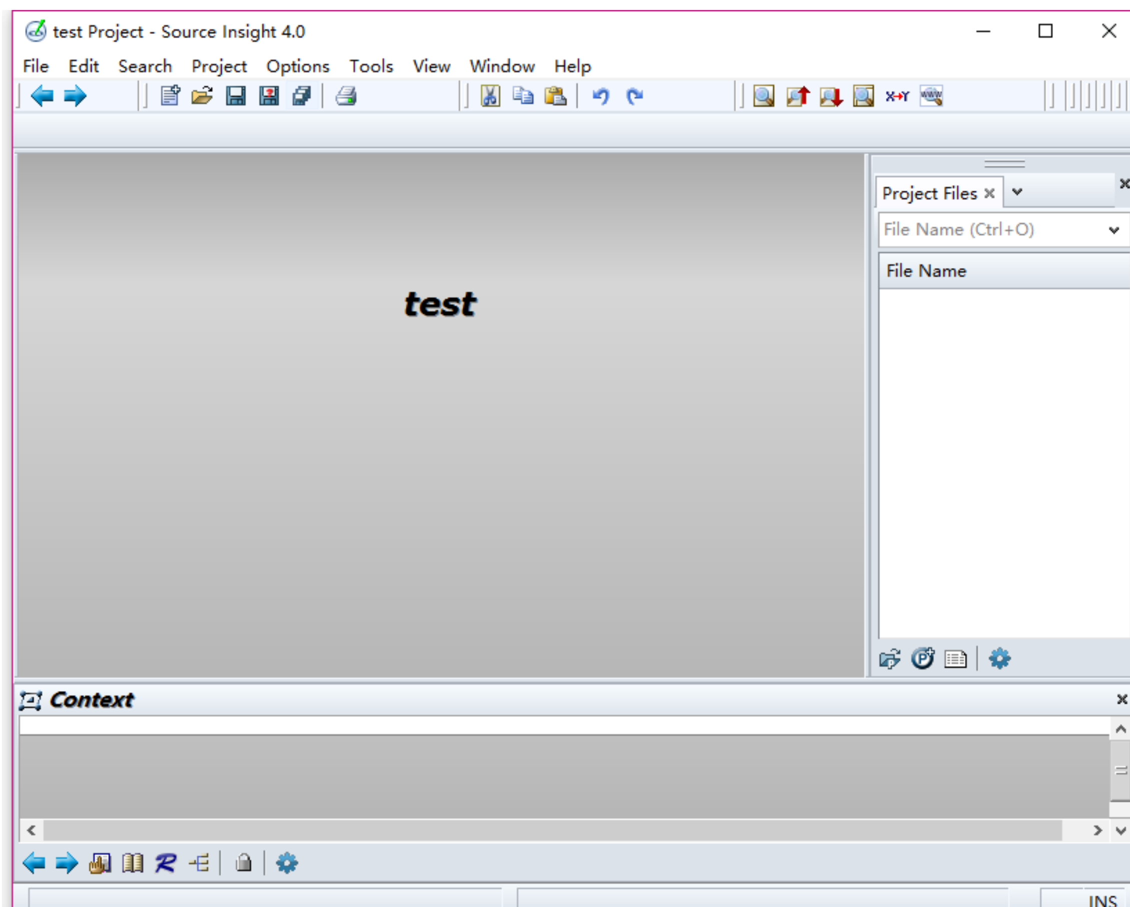


图 4.4.2.6 工程创建成功

我们发现图 4.4.2.6 好像和没有新建工程的界面没有区别？那是因为我们新建的工程是个空的工程，没有任何的源文件，所以看起来没啥变化。

2、新建源文件

我们在刚刚新建的工程里面新建两个文件：main.c 和 main.h，先新建 main.c 文件，点击：File->new，如图 4.4.2.7 所示：

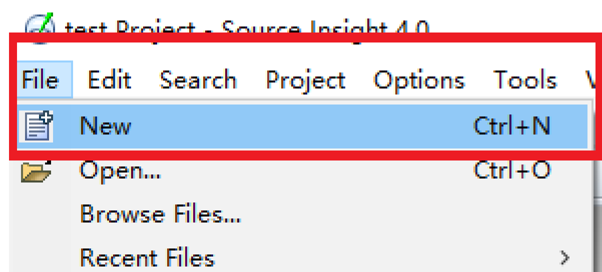


图 4.4.2.7 新建 c 文件

设置 c 文件的名字为 main.c，如图 4.4.2.8 所示：

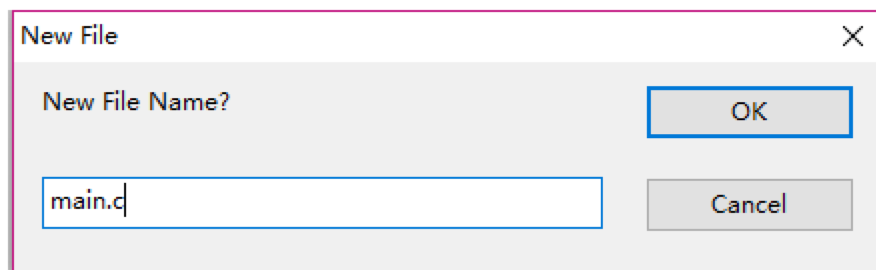


图 4.4.2.8 文件命名

文件命名完成以后点击“OK”按钮，文件创建完成，main.c 只是创建了但是还没有保存，更没有添加到我们的工程中，所以我们点击：File->Save，或者直接按下键盘上的“Ctrl+S”键来保存，保存界面如图 4.4.2.9 所示：

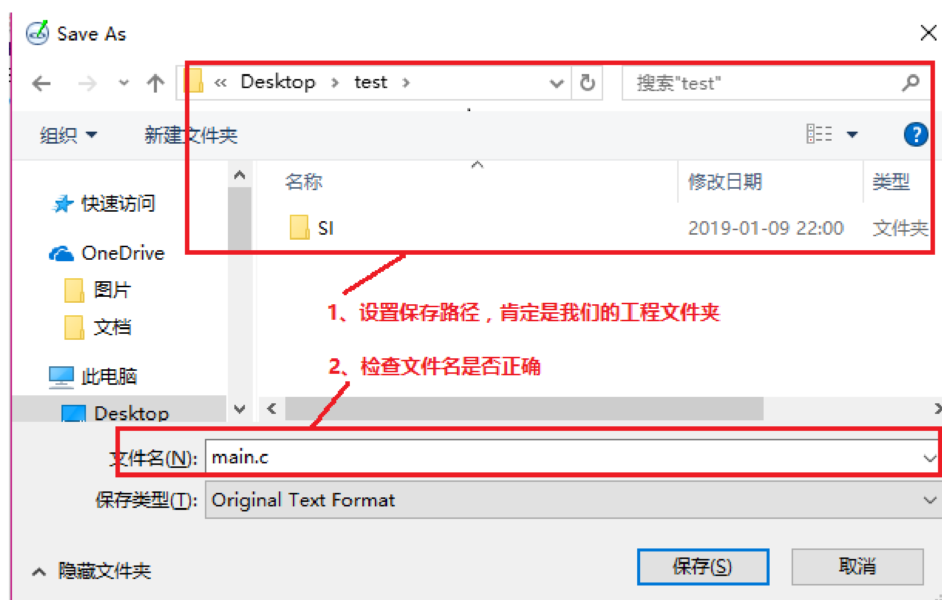


图 4.4.2.9 保存界面

设置好图 4.4.2.9 中的保存路径以后点击“保存”按钮即可，保存以后会弹出一个对话框，询问你是否要将刚刚保存的 C 文件添加到工程中，如图 4.4.2.10 所示：

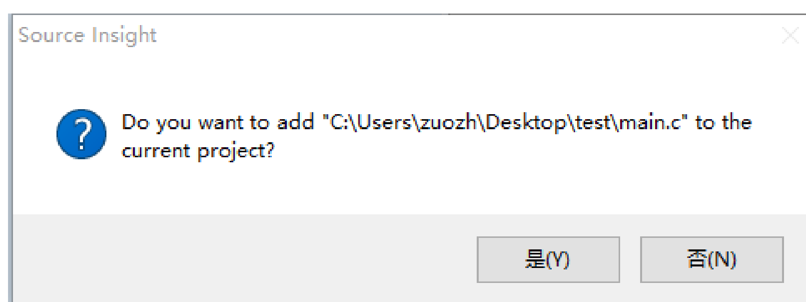


图 4.4.2.10 是否将文件添加到工程中。

我们肯定要选择“是”了，要将 main.c 添加到工程中的，添加完成以后的 Source Insight 界面如图 4.4.2.11 所示：

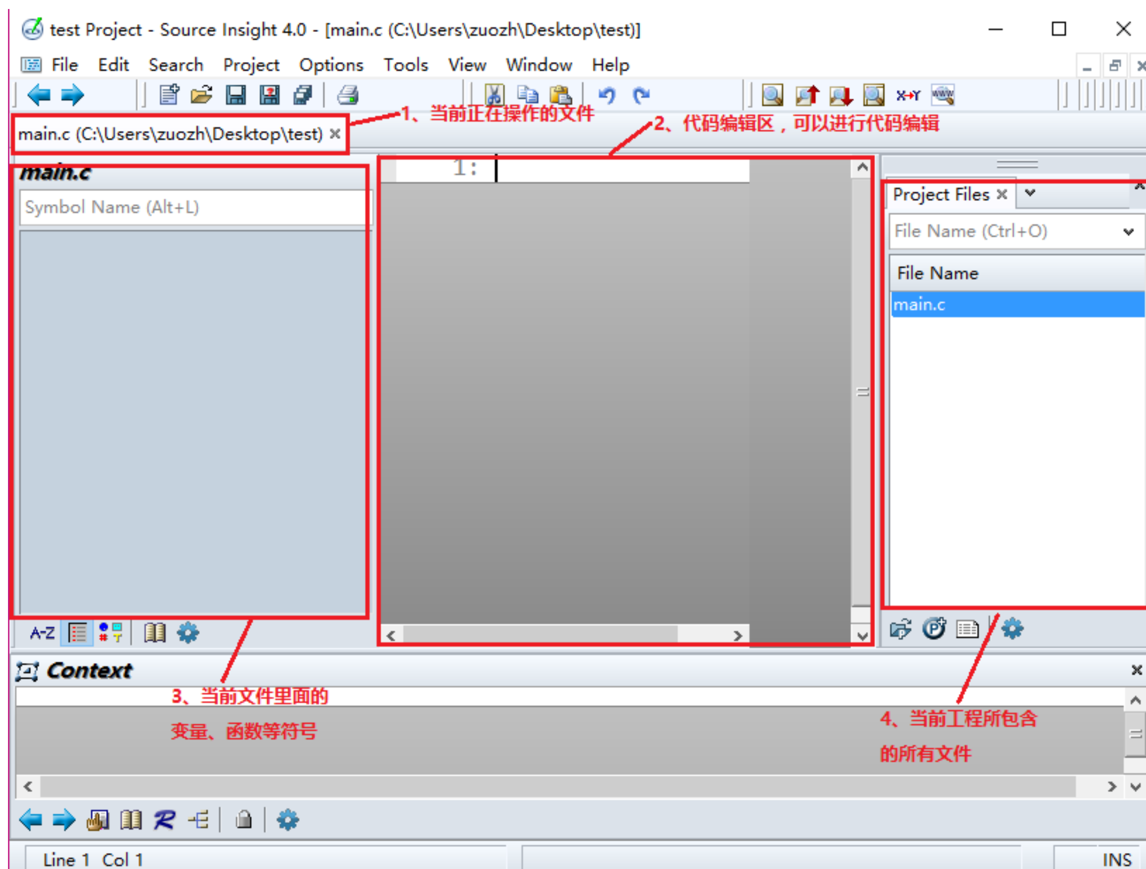


图 4.4.2.11 工程界面讲解

在图 4.4.2.11 中可以看到我们正在操作 main.c 这个文件，当前工程只有 main.c 这一个文件，中间部分就是我们的代码编辑区，我们可以在里面写代码。同样的方法我们在新建一个 main.h 头文件，

3、编写代码

我们在工程中创建了 main.c 和 main.h 两个源文件，接下来在这两个文件中编写代码，在 main.c 和 main.h 中分别写入如下代码：

示例代码 4.4.2.1 main.c 文件代码

```
1  #include "main.h"
2  #include "stdio.h"
3
4  void main(int argc, char *argv[])
5  {
6   printf("this is a test file");
7  }
8
```

示例代码 4.4.2.2 main.h 文件代码

```
9  #ifndef _MAIN_H
10 #define _MAIN_H
11
12
13 #endif
```

编写完成以后 Source Insight 界面如图 4.4.2.12 所示：

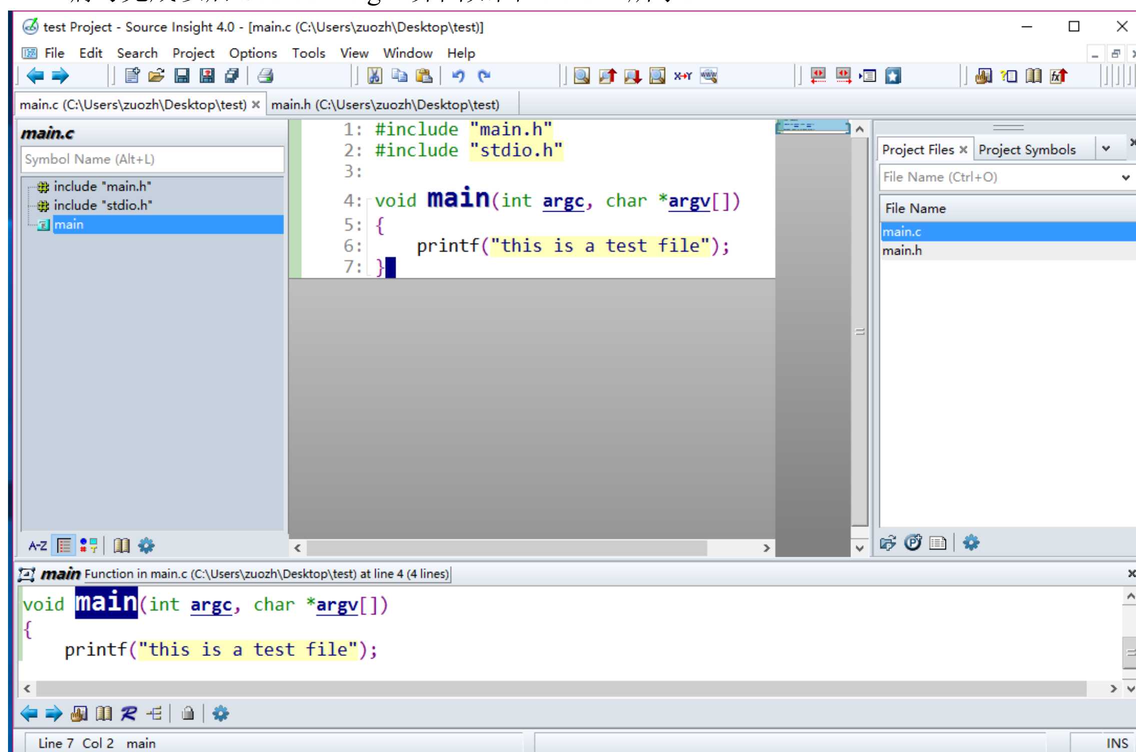


图 4.4.2.12 编写代码后的工程

4、工程同步

代码编写完成以后需要对 Source Insight 做一次同步操作，同步的目的是为了可以进行函数跟踪，比如 MDK 中直接跳转到某个函数的定义处查看函数源码。同步的方法很简单，点击 Project->Synchronize Files，如图 4.4.2.13 所示：

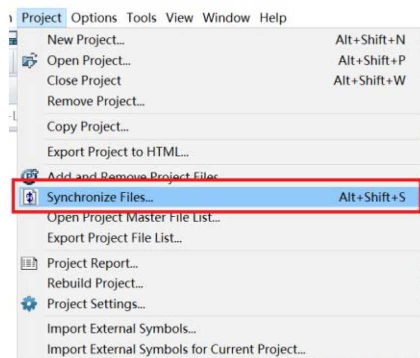


图 4.4.2.13 工程同步

点击“Synchronize Files”以后打开同步对话框，如图 4.4.2.14 所示：

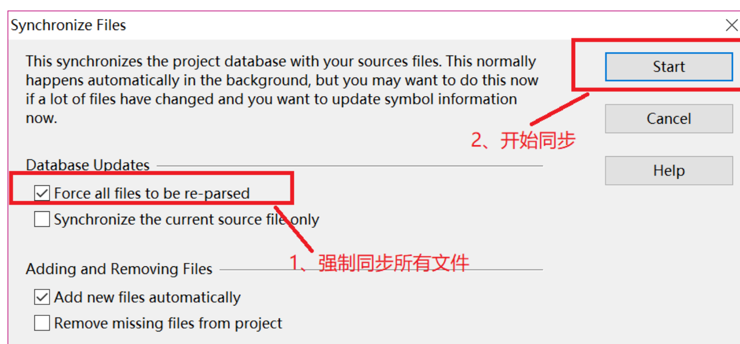


图 4.4.2.14 同步设置

按照图 4.4.2.14 所示设置同步，设置好以后点击“Start”开始同步，等待同步完成，如果工程很小的话同步速度会很快！可能看不到同步的过程，如果工程比较大的话同步就会多花一点时间。

关于 Source Insight 的安装以及使用就讲解到这里，大家自行多练习几遍 Source Insight 创建工程和新建文件操作。

4.4.3 Source Insight 解决中文乱码

第一次装好 Source Insight，如果打开有中文的文件，可能中文显示会乱码，如图 4.4.3.1 所示：

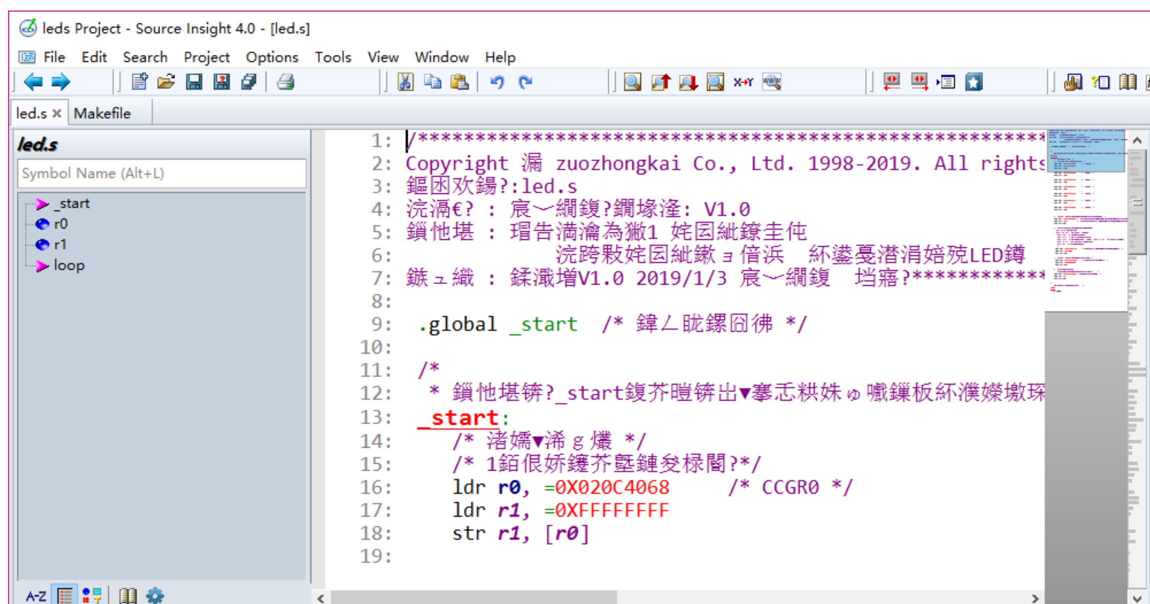


图 4.4.3.1 中文乱码

这是因为编码方式没有选对，点击 Options->Preferences…，如图 4.4.3.2 所示：

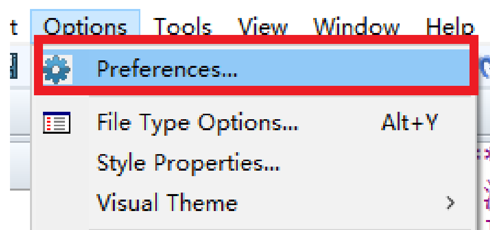


图 4.4.3.2 Preferences 对话框打开方式

打开以后按照图 4.4.3.3 所示设置：

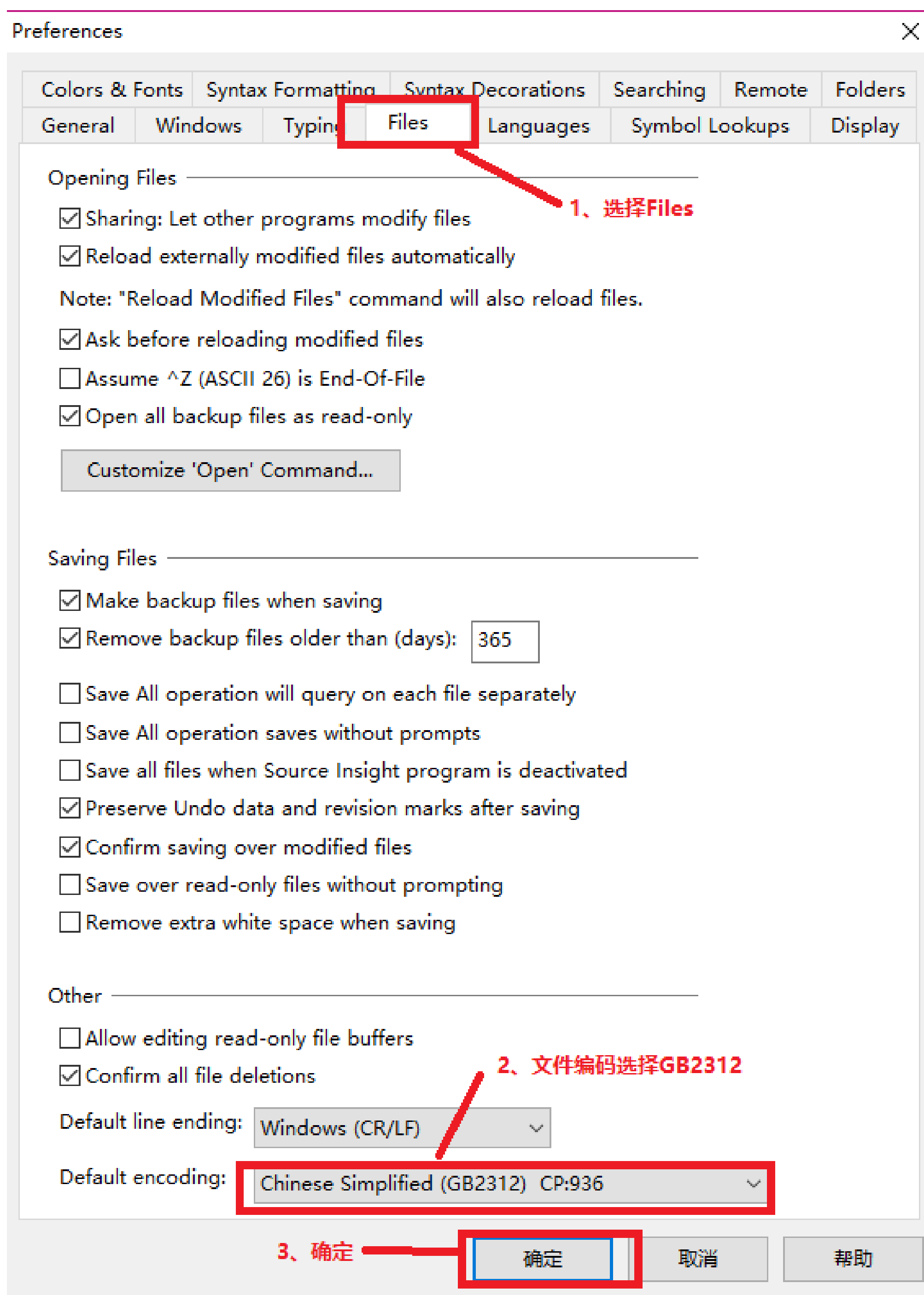


图 4.4.3.3 文件编码设置

将文件编码改为 GB2312 以后中文显示就正常了，如果中文还是显示乱码的话那就试着将图 4.4.3.3 中的“Default line ending”改为“Unix(LF)”，将“Default encoding”改为“UTF8”，如图 4.4.3.4 所示：

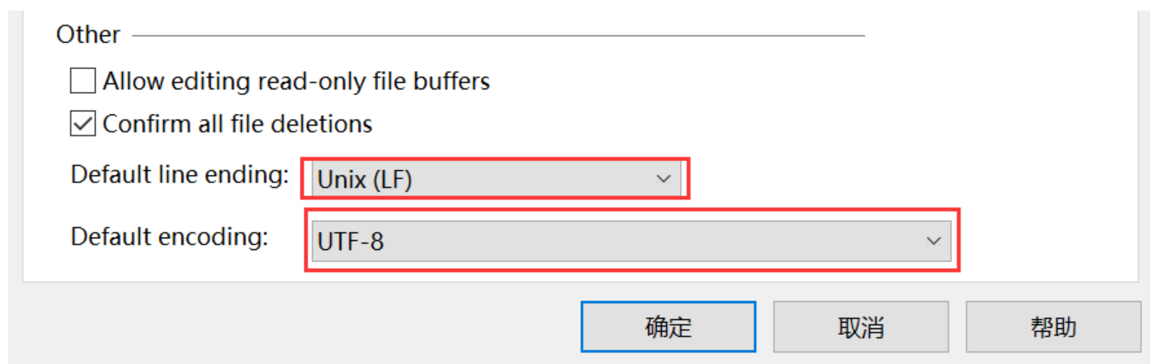


图 4.4.3.4 改为 UTF-8 编码

这是因为 Linux 下是 UTF-8 编码的，如果你的工程是从 Linux 下拷贝出来的，那么肯定就要使用 UTF8 编码才能正常显示。中文正常显示如图 4.4.3.5 所示：

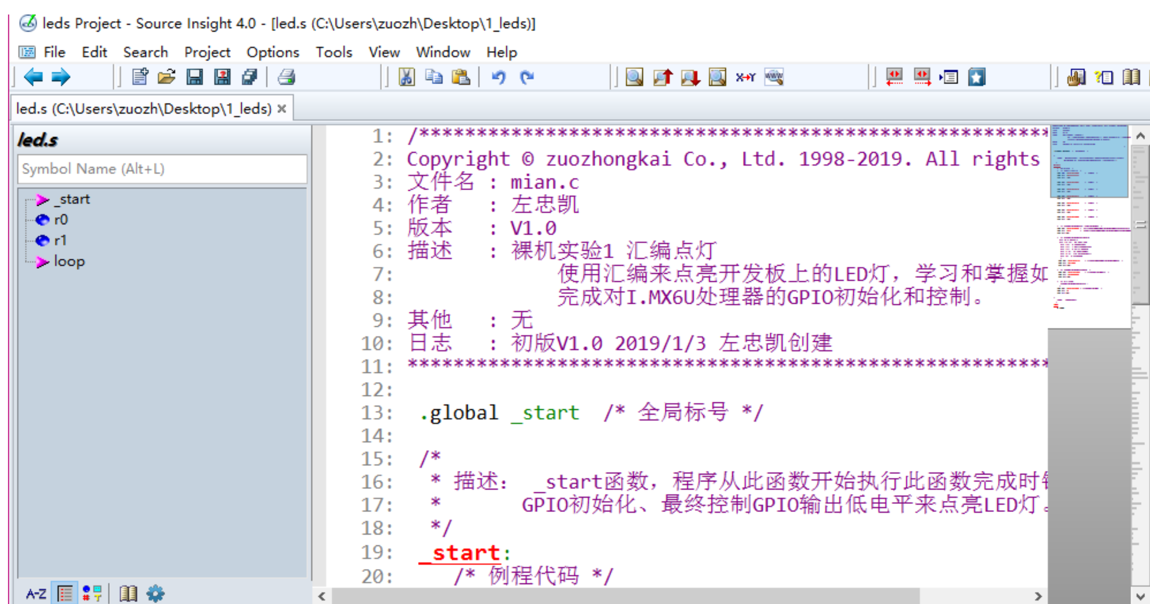


图 4.4.3.5 中文显示正常

4.5 Visual Studio Code 软件的安装和使用

4.5.1 Visual Studio Code 的安装

Visual Stuio Code 和 Source Insight 一样，都是编辑器，Visual Studio Sode 本教程以后就简称为 VSCode，VSCode 是微软出的一款编辑器，但是免费的。VSCode 有 Windows、Linux 和 macOS 三个版本的，是一个跨平台的编辑器。VSCode 下载地址是：https://code.visualstudio.com/，下载界面如图 4.5.1.1 所示：

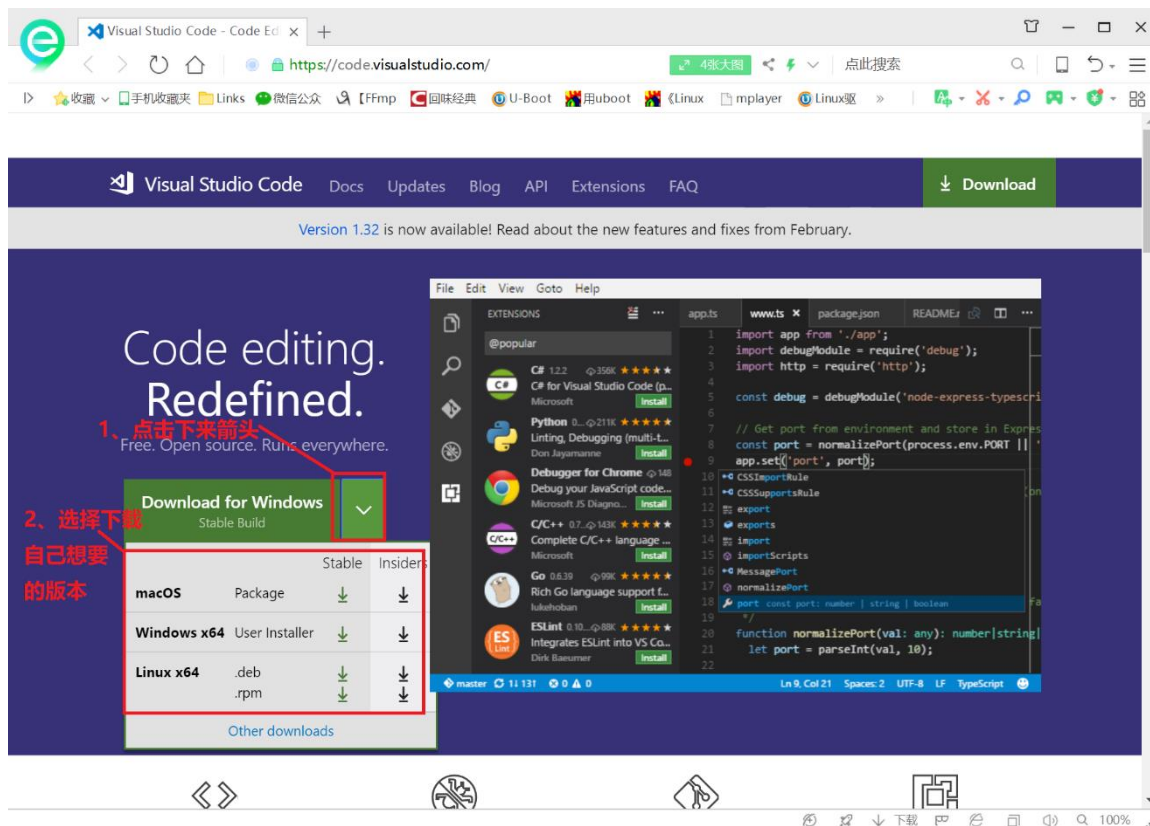


图 4.5.1.1 VSCode 下载界面

在图 4.5.1.1 中下载自己想要的版本，本教程需要 Windows 和 Linux 这两个版本，所以下载这两个即可，我们已经下载好并放入了开发板光盘中，路径为：3、软件->Visual Studio Code。

1、Windows 版本安装

Windows 版本的安装和容易，和其他 Windows 一样，双击.exe 安装包，然后一路“下一步”即可，安装完成以后在桌面上就会有 VSCode 的图标，如图 4.5.1.2 所示：

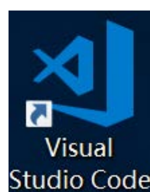


图 4.5.1.2 VSCode 图标

双击图 4.5.1.2 打开 VSCode，默认界面如图 4.5.1.3 所示：

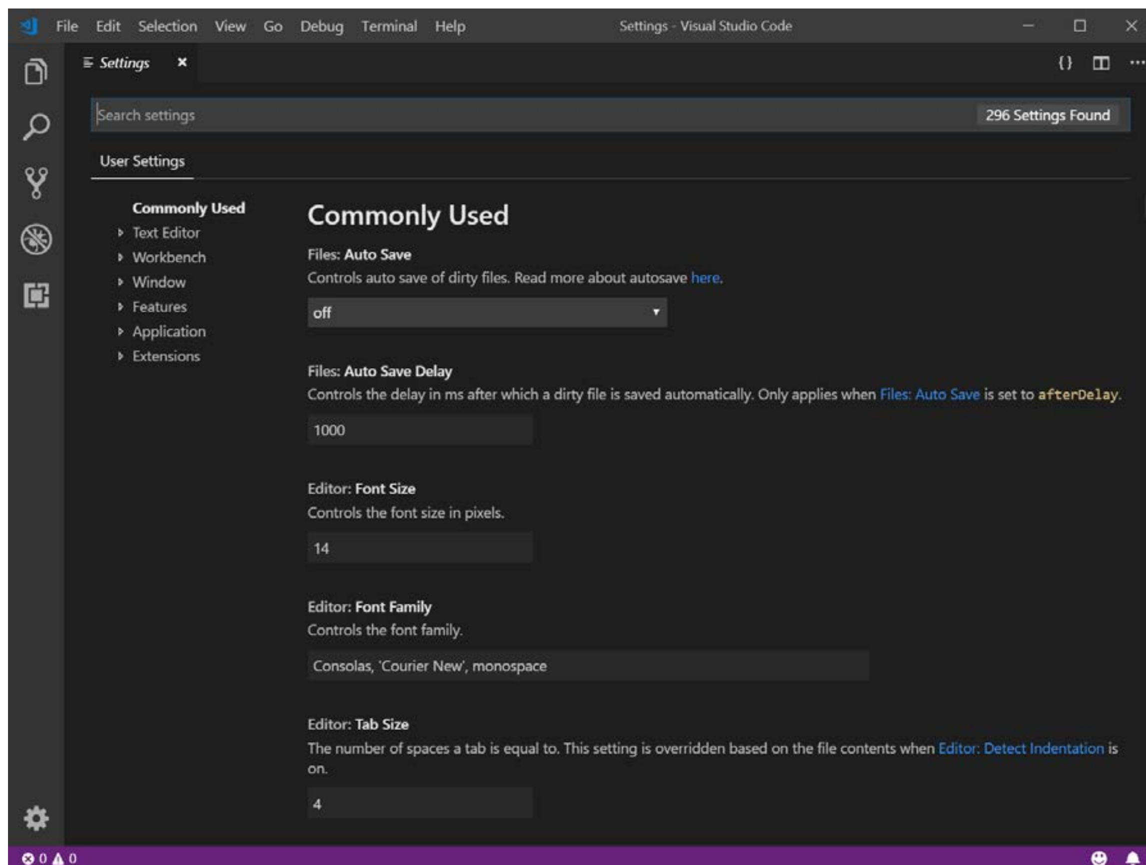


图 4.5.1.3 VSCode 默认界面

2、Linux 版本安装

我们有时候也需要在 Ubuntu 下阅读代码，所以还需要在 Ubuntu 下安装 VSCode。Linux 下的 VSCode 安装包我们也放到了开发板光盘中， 将开发板光盘中的.deb 软件包拷贝到 Ubuntu 系统中，然后使用如下命令安装：

```
sudo dpkg -i code_1.35.3-1552606978_amd64.deb
```

等待安装完成，如图 4.5.1.4 所示：

```
zuozhongkai@ubuntu:~/linux/tool$ sudo dpkg -i code_1.32.3-1552606978_amd64.deb
正在选中未选择的软件包 code。
(正在读取数据库 ... 系统当前共安装有 183257 个文件和目录。)
正准备解包 code_1.32.3-1552606978_amd64.deb  ...
正在解包 code (1.32.3-1552606978) ...
正在设置 code (1.32.3-1552606978) ...
正在处理用于 gnome-menus (3.13.3-6ubuntu3.1) 的触发器 ...
正在处理用于 desktop-file-utils (0.22-1ubuntu5.1) 的触发器 ...
正在处理用于 bamfdaemon (0.5.3~bzr0+16.04.20160824-0ubuntu1) 的触发器 ...
Rebuilding /usr/share/applications/bamf-2.index...
正在处理用于 mime-support (3.59ubuntu1) 的触发器 ...
zuozhongkai@ubuntu:~/linux/tool$
```

图 4.5.1.4 VSCode 安装过程

安装完成以后搜索“Visual Studio Code”就可以找到，如图 4.5.1.5 所示：

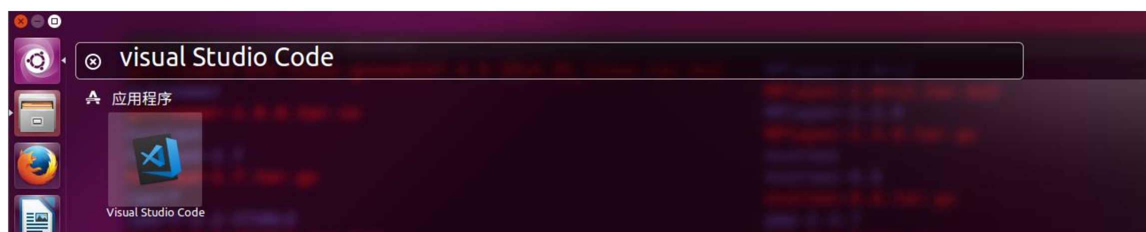


图 4.5.1.5 Visual Studio Code

每次打开 VSCode 都要搜索，太麻烦了，我们可以将图标添加到 Ubuntu 桌面上，安装的所有软件图标都在目录/usr/share/applications 中，如图 4.5.1.6 所示：



图 4.5.1.6 软件图标

在图 4.5.1.6 中找到 Visual Studio Code 的图标，然后点击鼠标右键，选择复制到->桌面，如图 4.5.1.7 所示：

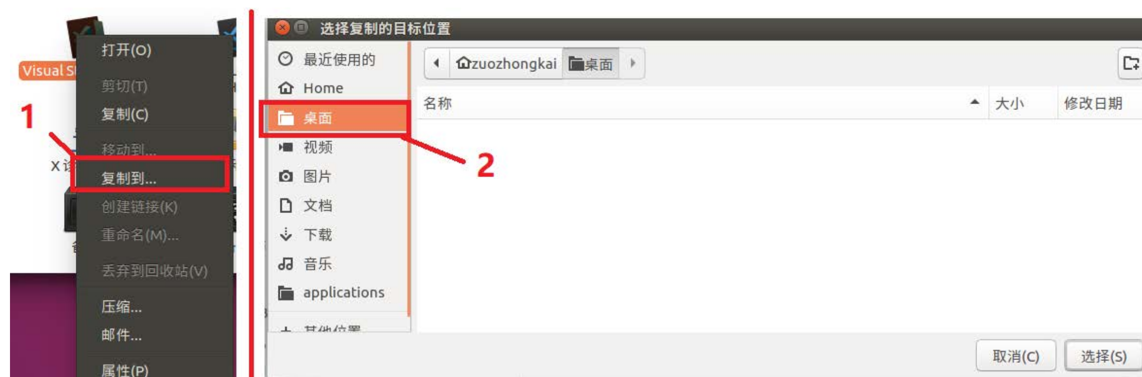


图 4.5.1.7 复制图标到桌面

按照图 4.5.1.7 所示方法将 VSCode 图标复制到桌面，以后直接双击图标即可打开 VSC，Ubuntu 下的 VSCode 打开以后如图 4.5.1.8 所示：

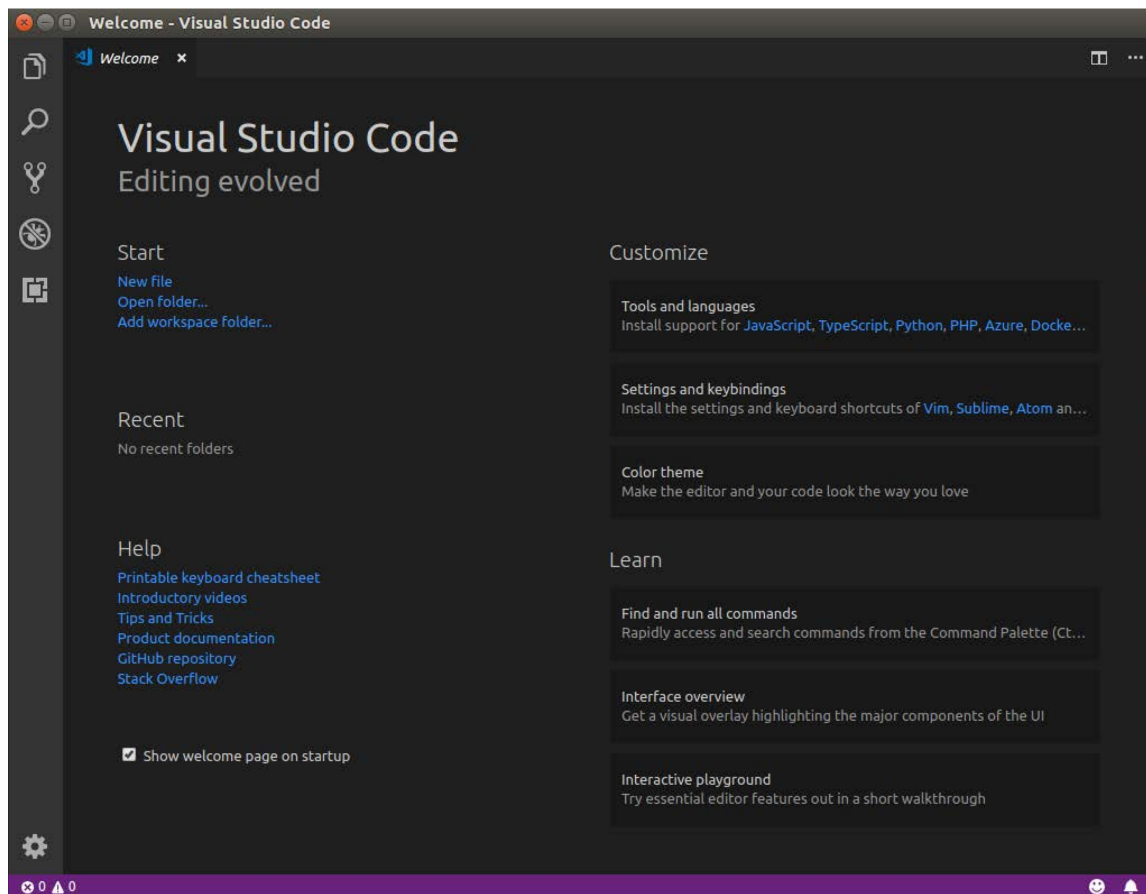


图 4.5.1.8 Linux 下的 VSCode

可以看出 Linux 下的 VSCode 和 Windows 下的基本是一样的，所以使用方法也是一样的。

4.5.2 Visual Studio Code 插件的安装

VSCode 支持多种语言，比如 C/C++、Python、C#等等，本教程我们主要用来编写 C/C++程序的，所以需要安装 C/C++的扩展包，扩展包安装很简单，如图 4.5.2.1 所示：

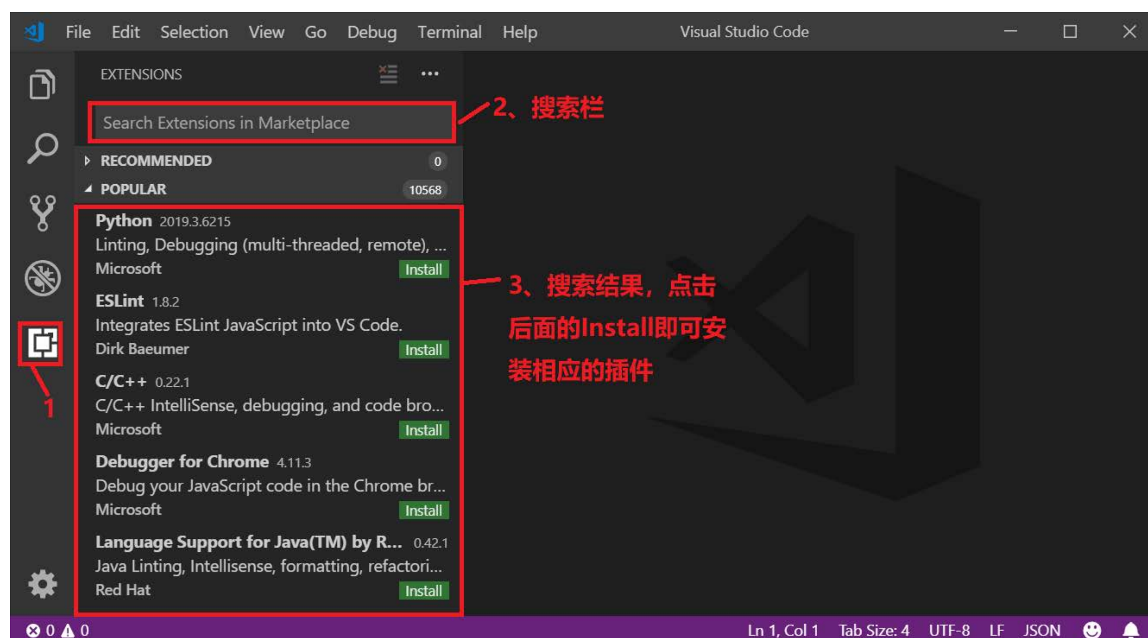


图 4.5.2.1 VSCode 插件安装

我们需要按照的插件有下面几个：

1)、C/C++，这个肯定是必须的。

2)、C/C++ Snippets，即 C/C++重用代码块。

3)、C/C++ Advanced Lint,即 C/C++静态检测 。

4)、Code Runner，即代码运行。

5)、Include AutoComplete，即自动头文件包含。

6)、Rainbow Brackets，彩虹花括号，有助于阅读代码。

7)、One Dark Pro，VSCode 的主题。

8)、GBKtoUTF8，将 GBK 转换为 UTF8。

9)、ARM，即支持 ARM 汇编语法高亮显示。

10)、Chinese(Simplified)，即中文环境。

11)、vscode-icons，VSCode 图标插件，主要是资源管理器下各个文件夹的图标。

12)、compareit，比较插件，可以用于比较两个文件的差异。

13)、DeviceTree，设备树语法插件。

如果要查看已经安装好的插件，可以按照图 4.5.2.2 所示方法查看：

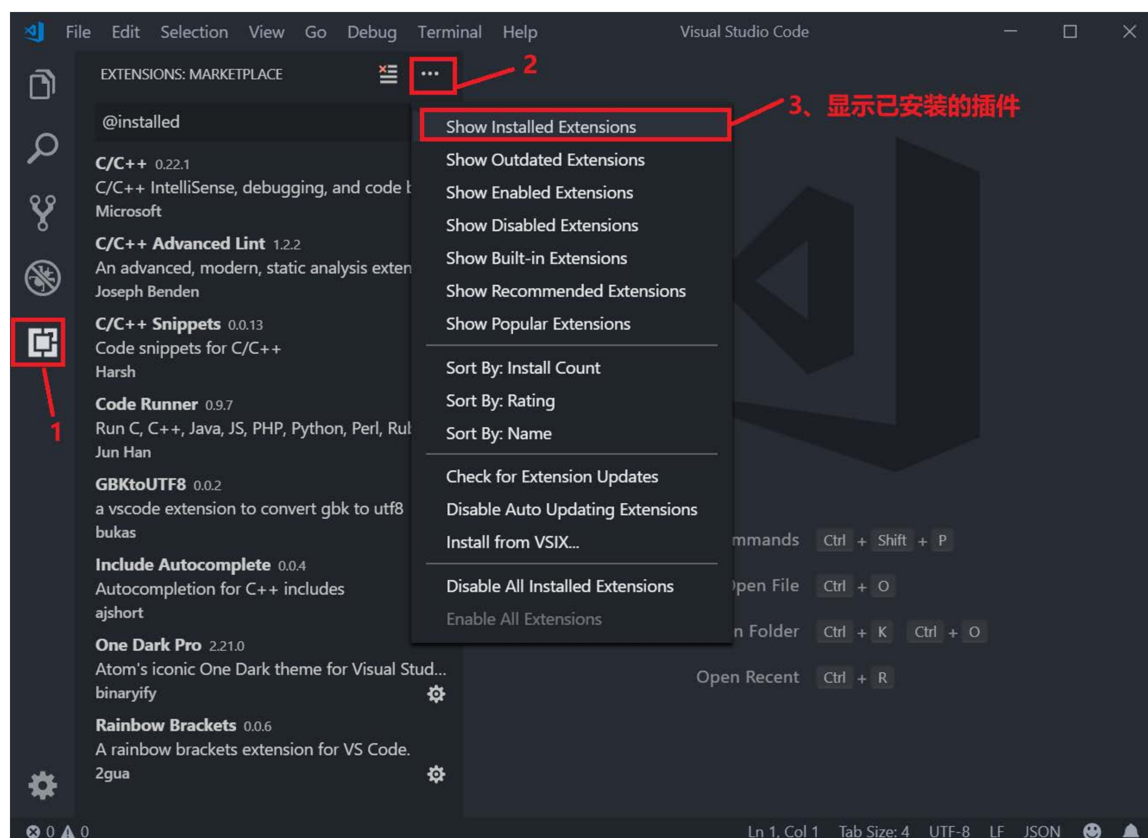


图 4.5.2.2 显示已安装的插件

安装好插件以后就可以进行代码编辑了，截至目前，VSCode 界面都是英文环境，我们已经安装了中文插件了，最后将 VSCode 改为中文环境，使用方法如图 4.5.2.3 所示：



适用于 VS Code 的中文（简体）语言包

此中文（简体）语言包为 VS Code 提供本地化界面。

使用方法

安装后，在 `locale.json` 中添加 `"locale": "zh-cn"`，即可载入中文（简体）语言包。要修改 `locale.json`，你可以同时按下 `Ctrl+Shift+P` 打开**命令面板**，之后输入 "config" 筛选可用命令列表，最后选择**配置语言**命令。请参阅文档并获取更多信息。

图 4.5.2.3 中文语言包使用方法

根据图 4.5.2.3 的提示，按下“Ctrl+Shift+P”打开搜索框，在搜索框里面输入“config”，然后选择“Configure Display Language”，如图 4.5.2.4 所示：

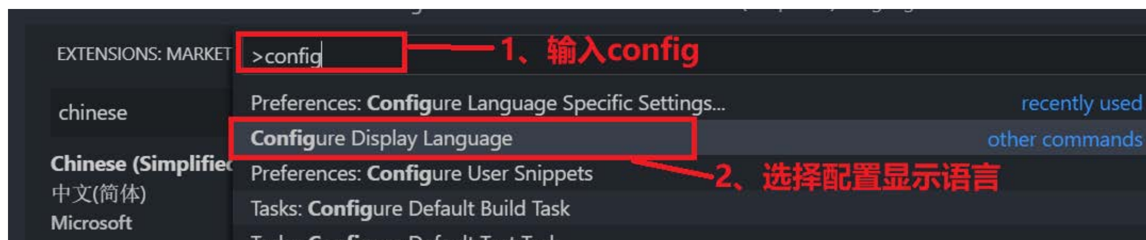


图 4.5.2.4 配置语言

在打开的 local.json 文件中将 locale 修改为 zh-cn，如图 4.5.2.5 所示：



图 4.5.2.5 修改 locale 变量

修改完成以后保存 local.json，然后重新打开 VSCode，测试 VSCode 就变成了中文的了，如图 4.5.2.6 所示：



图 4.5.2.6 中文环境

4.5.3 Visual Studio Code 新建工程

新建一个文件夹用于存放工程，比如我新建了文件夹目录为 E:\VScode_Program\1_test，路径尽量不要有中文和空格打开 VSCode。然后在 VSCode 上点击文件->打开文件夹…，选刚刚创建的“1_test”文件夹，打开以后如图 4.5.3.1 所示：



图 4.5.3.1 打开的文件夹

从图 4.5.3.1 可以看出此时的文件夹“1_TEST”是空的，点击文件->将工作区另存为…，打开工作区命名对话框，输入要保存的工作区路径和工作区名字，如图 4.5.3.2 所示：

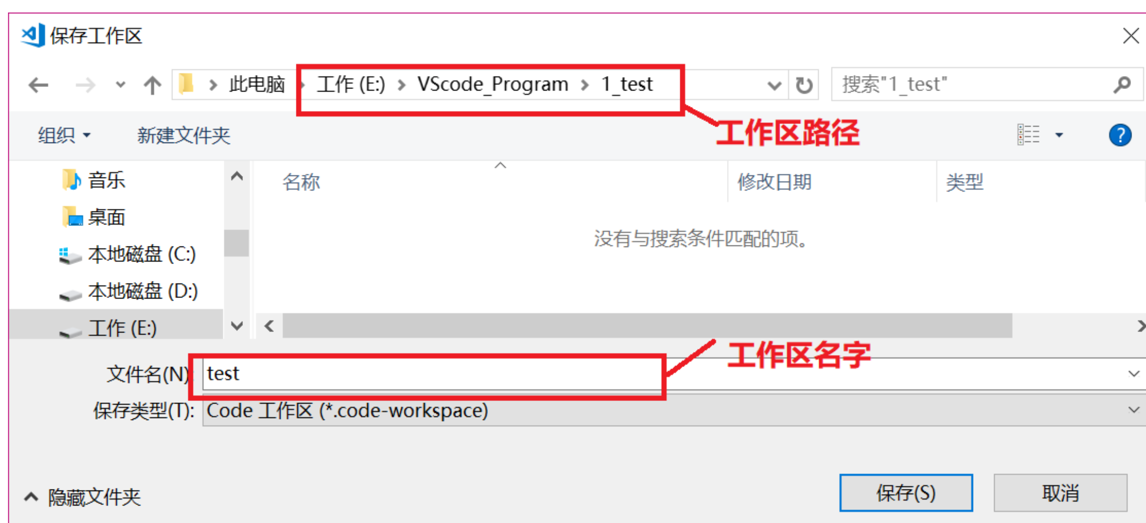


图 4.5.3.2 工作区保存设置

工作区保存成功以后，点击图 4.5.3.1 中的“新建文件”按钮创建 main.c 和 main.h 这两个文件，创建成功以后 VSCode 如图 4.5.3.3 所示：

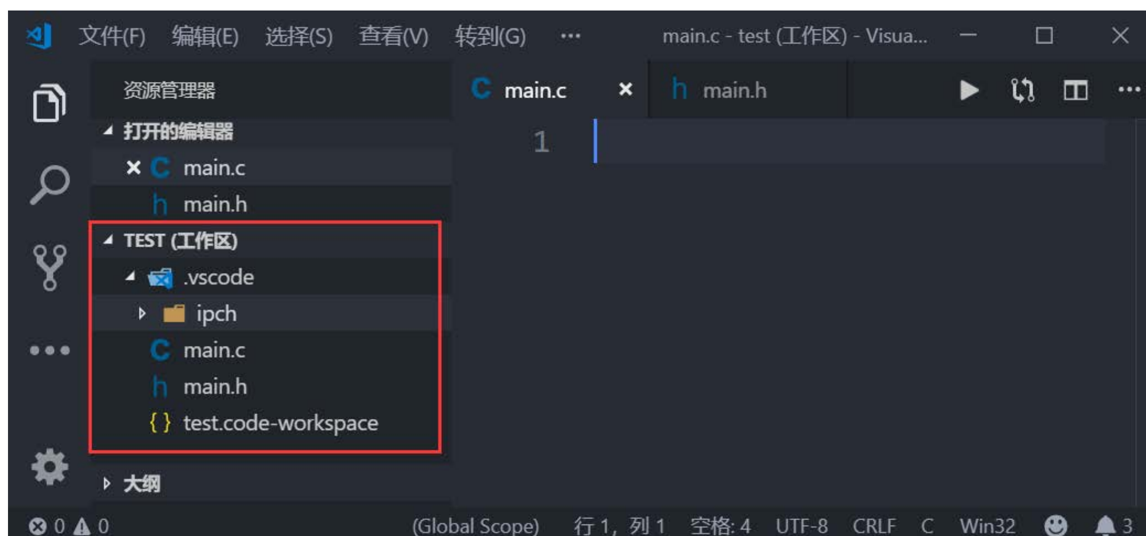


图 4.5.3.2 新建文件以后的 VSCode

从图 4.5.3.2 可以看出，此时“实验 1 TEST”中有.vscode 文件夹、mian.c 和 mian.h，这三个文件和文件夹同样会出现在“实验 1 test”文件夹中，如图 4.5.3.3 所示：

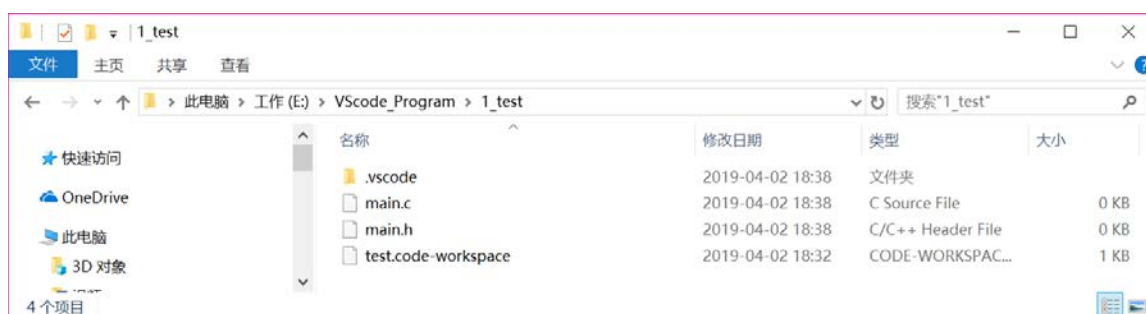


图 4.5.3.3 实验文件夹

在 main.h 中输入如下所示内容：

示例代码 4.5.3.1 main.h 文件代码

```
1 #include <stdio.h>
2
3 int add(int a, int b);
```

在 main.c 中输入如下所示内容：

示例代码 4.5.3.2 main.c 文件代码

```
1  #include <main.h>
2
3  int add(int a, int b)
4  {
5      return (a + b);
6  }
7
8  int main(void)
9  {
10     int value = 0;
11
```

```
12     value = add(5, 6);
13     printf("5 + 6 = %d", value);
14     return 0;
15 }
```

代码编辑完成以后 VSCode 界面如图 4.5.3.4 所示：

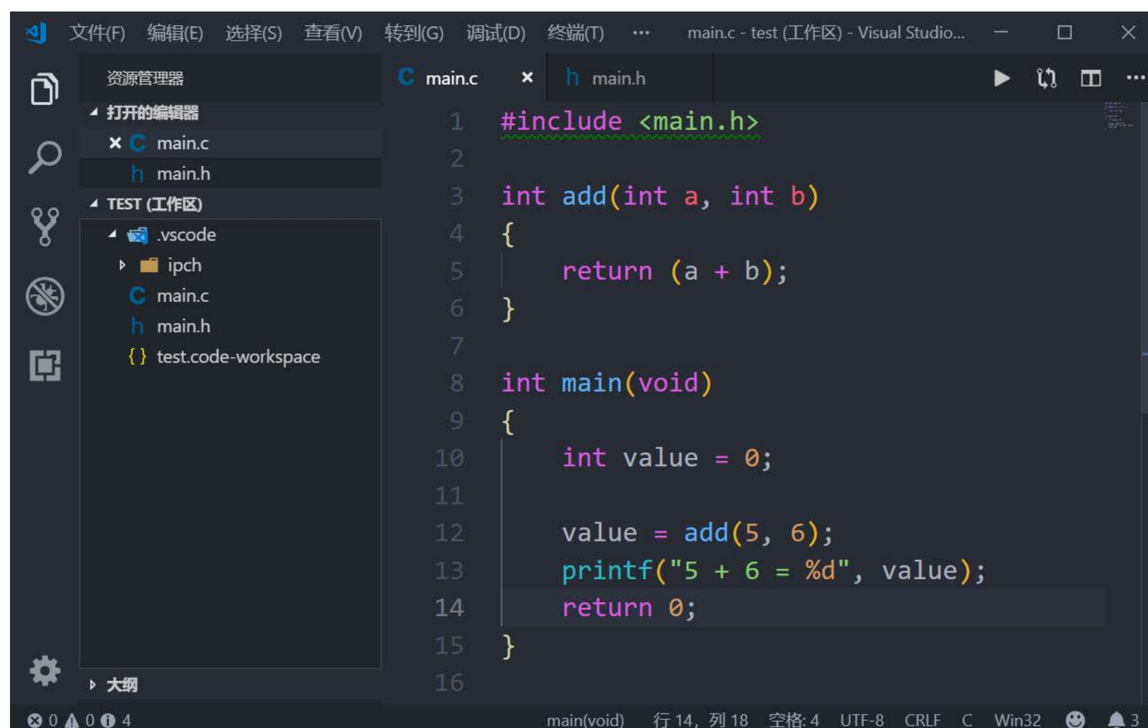


图 4.5.3.4 代码编辑完成以后的界面

从图 4.5.3.4 可以看出，VSCode 的编辑的代码高亮很漂亮，阅读起来很舒服。但是此时提示找不到“stdio.h”这个头文件，如图 4.5.3.5 所示错误提示：

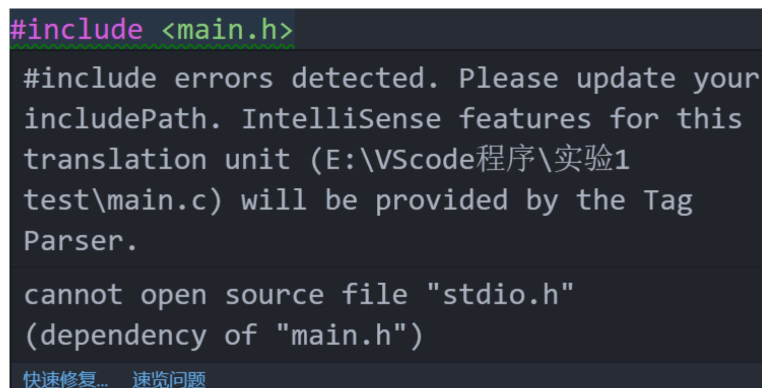


图 4.5.3.5 头文件找不到。

图 4.5.3.5 中提示找不到“main.h”，同样的在 main.h 文件中会提示找不到“stdio.h”。这是因为我们没有添加头文件路径。按下“Ctrl+Shift+P”打开搜索框，然后输入“Edit configurations”，选择“C/C++:Edit configurations…”，如图 4.5.3.6 所示：

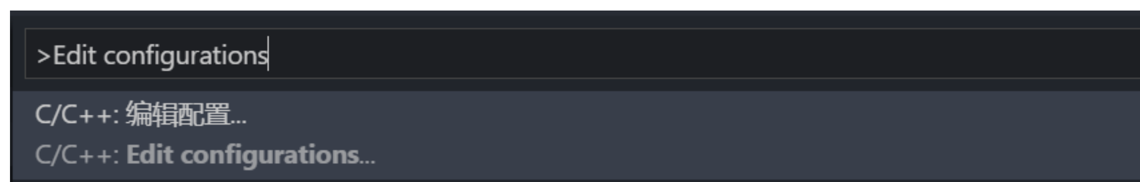


图 4.5.3.6 打开 C/C++编辑配置文件

C/C++的配置文件是个 json 文件，名为：c_cpp_properties.json，此文件默认内容如图 4.5.3.7 所示：

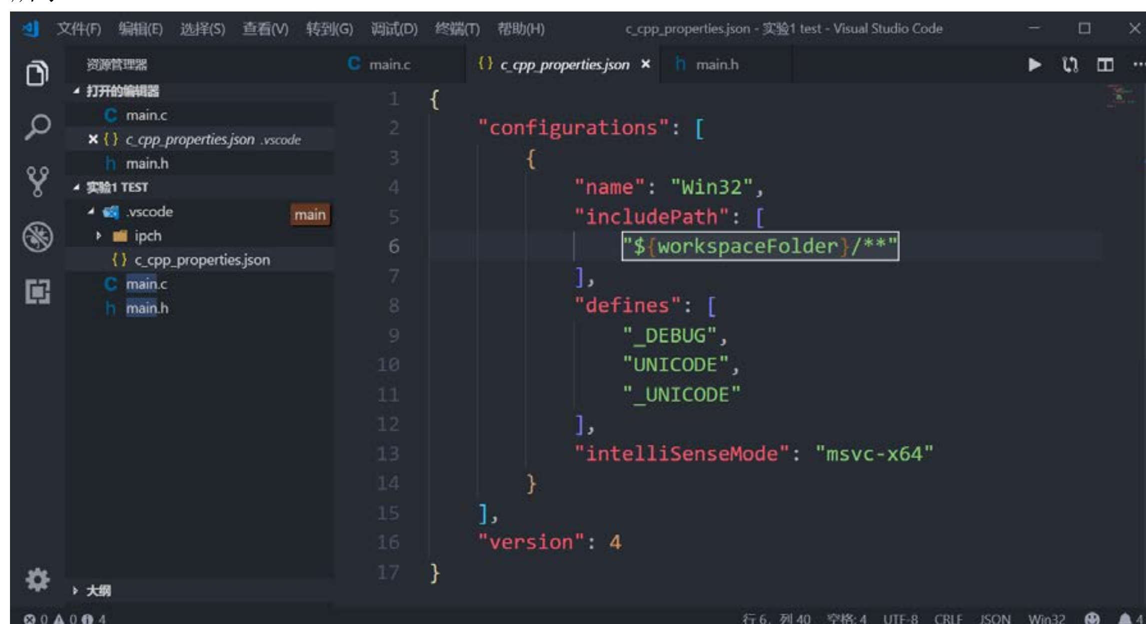


图 4.5.3.7 文件 c_cpp_properties.json 内容

c_cpp_properties.json 中的变量"includePath"用于指定工程中的头文件路径，但是"stdio.h"是 C 语言库文件，而 VSCode 只是个编辑器，没有编译器，所以肯定是没有 stdio.h 的，除非我们自行安装一个编译器，比如 CygWin，然后在 includePath 中添加编译器的头文件。这里我们就不添加了，因为我们不会使用 VSCode 来编译程序，这里主要知道如何指定头文件路径就可以了，后面有实际需要的时候再来讲。

我们在 VScode 上打开一个新文件的话会覆盖掉以前的文件，这是因为 VSCode 默认开启了预览模式，预览模式下单击左侧的文件就会覆盖掉当前的打开的文件。如果不想覆盖的话采用双击打开即可，或者设置 VSCode 关闭预览模式，设置如图 4.5.3.8 所示：

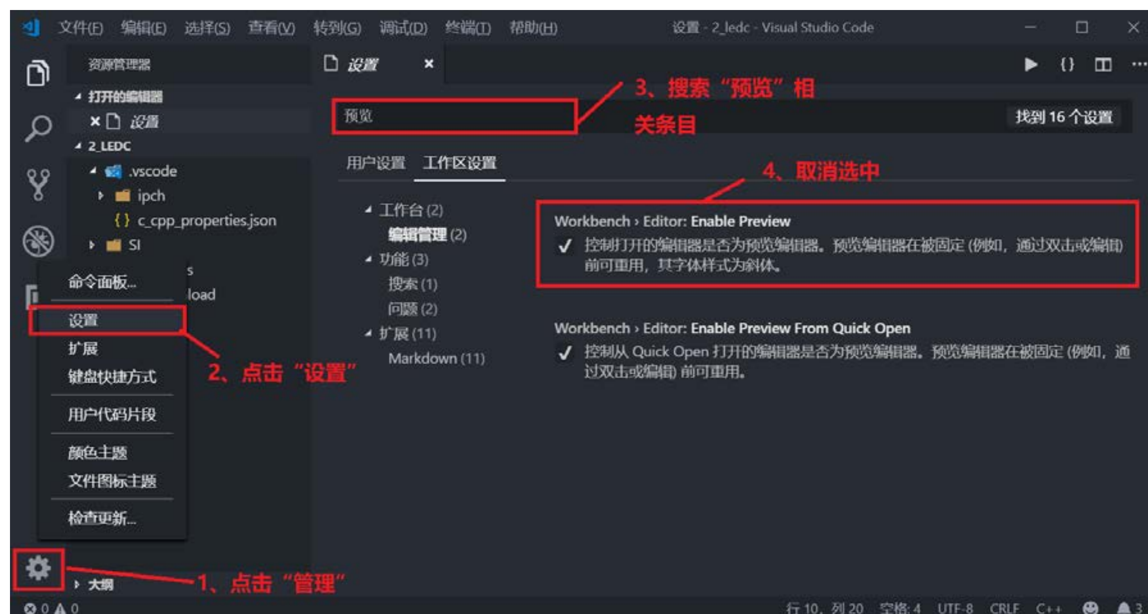


图 4.5.3.8 取消预览

我们在编写代码的时候有时候会在右下角有如图 4.5.3.9 所示的警告提示：

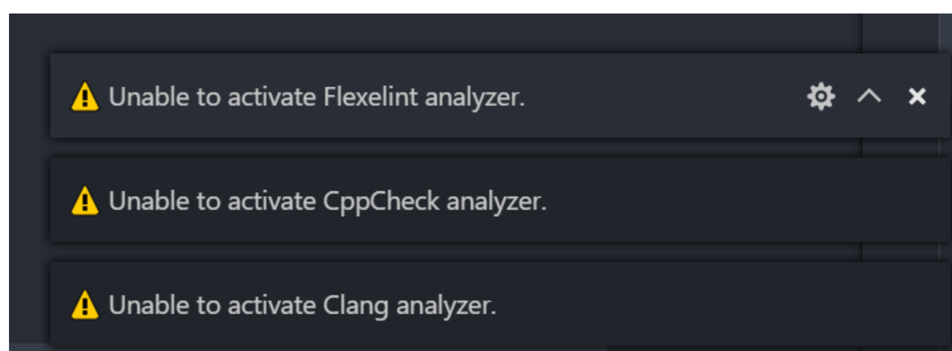


图 4.5.3.9 错误提示

这是因为插件 C/C++ Lint 打开了几个功能，我们将其关闭就可以了，顺便也可以学习一下 VSCode 插件配置方法，如图 4.5.3.10 所示：

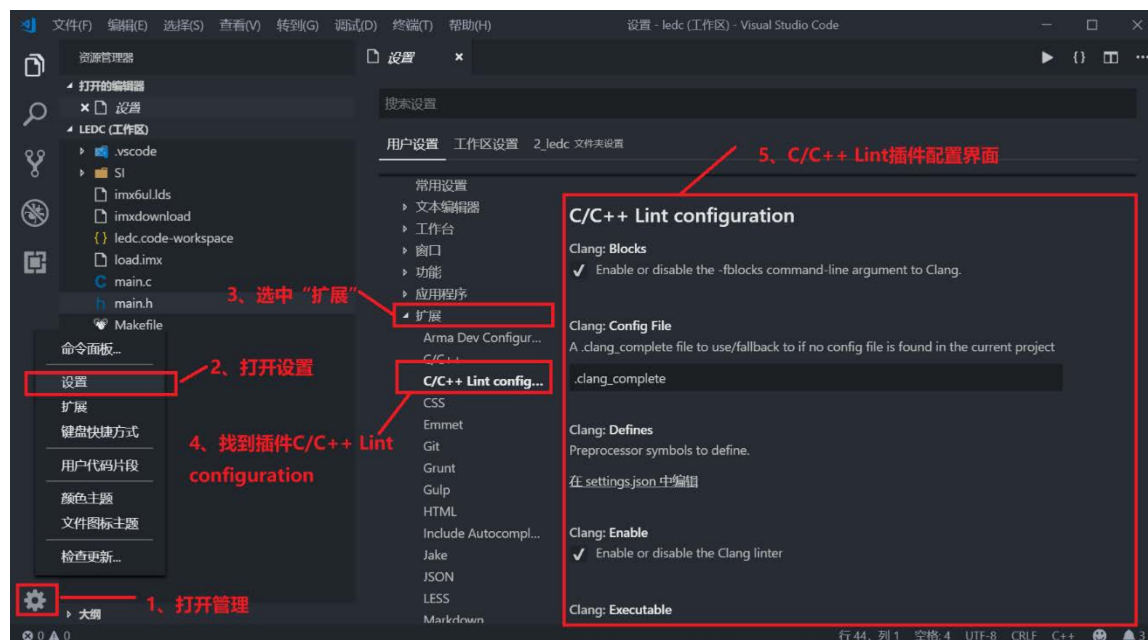


图 4.5.3.10 C/C++ Lint 配置界面

在 C/C++ Lint 配置界面上找到 CLang:Enable、Cppcheck:Enable、Flexlint:Enable 这个三个，然后取消掉勾选即可，如图 4.5.3.11 所示：

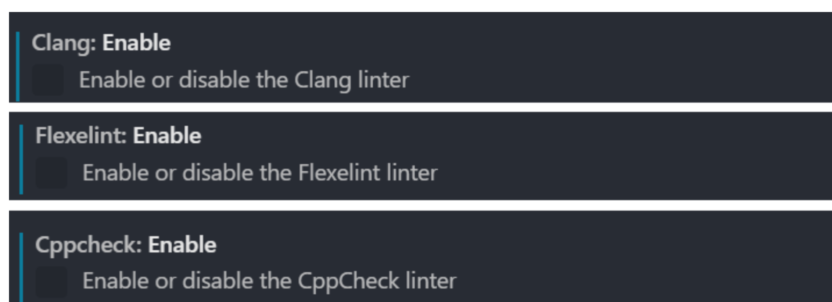


图 4.5.3.11 C/C++ Lint 配置

按照图 4.5.3.11 所示取消这三个有关 C/C++ Lint 的配置以后就不会有图 4.5.3.9 所示的错误提示了。但是关闭 Cppcheck:Enable 以后 VSCode 就不能实时检查错误了，大家根据实际情况选择即可。

4.6 CH340 串口驱动安装

我们一般在 Windwos 下通过串口来调试程序，或者使用串口作为终端，I.MX6U-ALPHA 开发板使用 CH340 这个芯片实现了 USB 转串口功能，CH340 是一枚江苏沁恒生产的国产芯片，稳定性还是很不错的，这里我们要多多支持国产嘛。

先通过 USB 线将开发板的串口和电脑连接起来起来，连接方式如图 4.6.1：

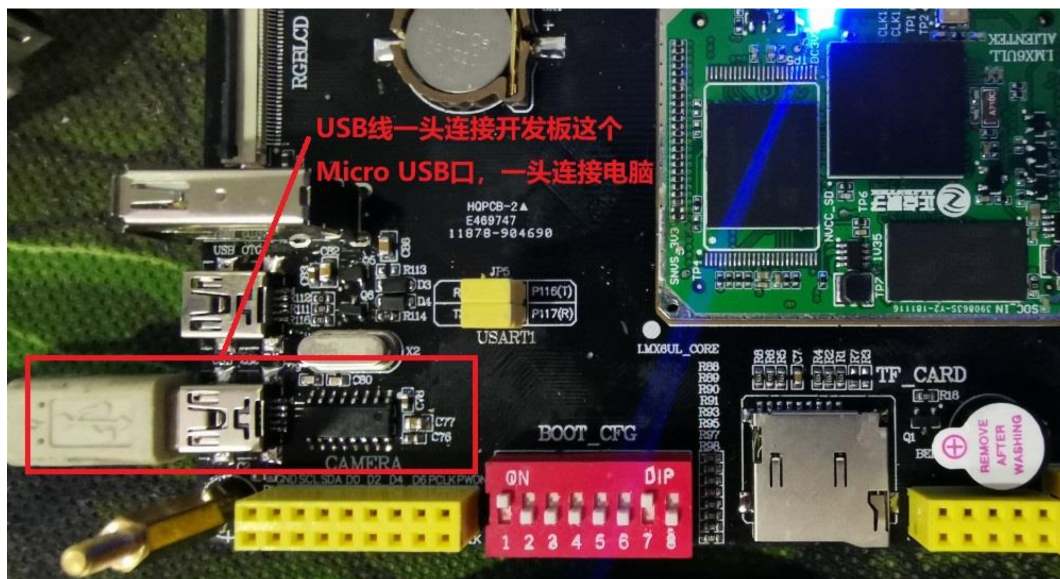


图 4.6.1 开发板串口连接方式

CH340 是需要安装驱动的，驱动我们已经放到了开发板光盘中，路径：开发板光盘->3、软件->CH340 驱动(USB 串口驱动)_XP_WIN7 共用->SETUP.EXE，，双击 SETUP.EXE，打开如图 4.6.2 所示安装界面：



图 4.6.2 CH340 驱动安装

点击图 4.6.2 中的“安装”按钮开始安装驱动，等待驱动安装完成，驱动安装完成以后会有如图 4.6.3 所示提示：

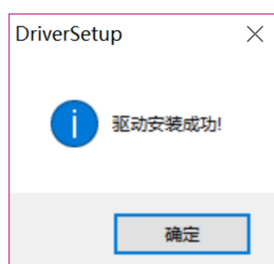


图 4.6.3 驱动安装成功

点击图 4.6.3 中的“确定”按钮退出安装，重新插拔一下串口线。打开设备管理器，打开方式是在 Windows 上的“此电脑”图标上点击鼠标右键，选择“管理”，如图 4.6.4

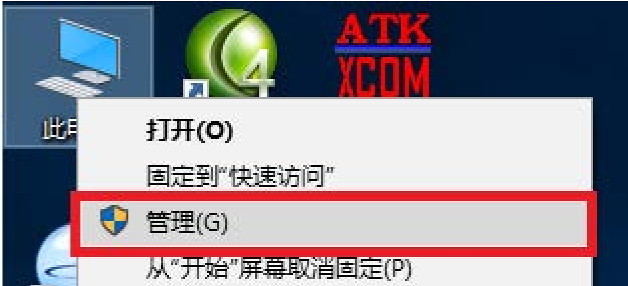


图 4.6.4 打开管理窗口

打开以后的计算机管理器如图 4.6.5 所示：

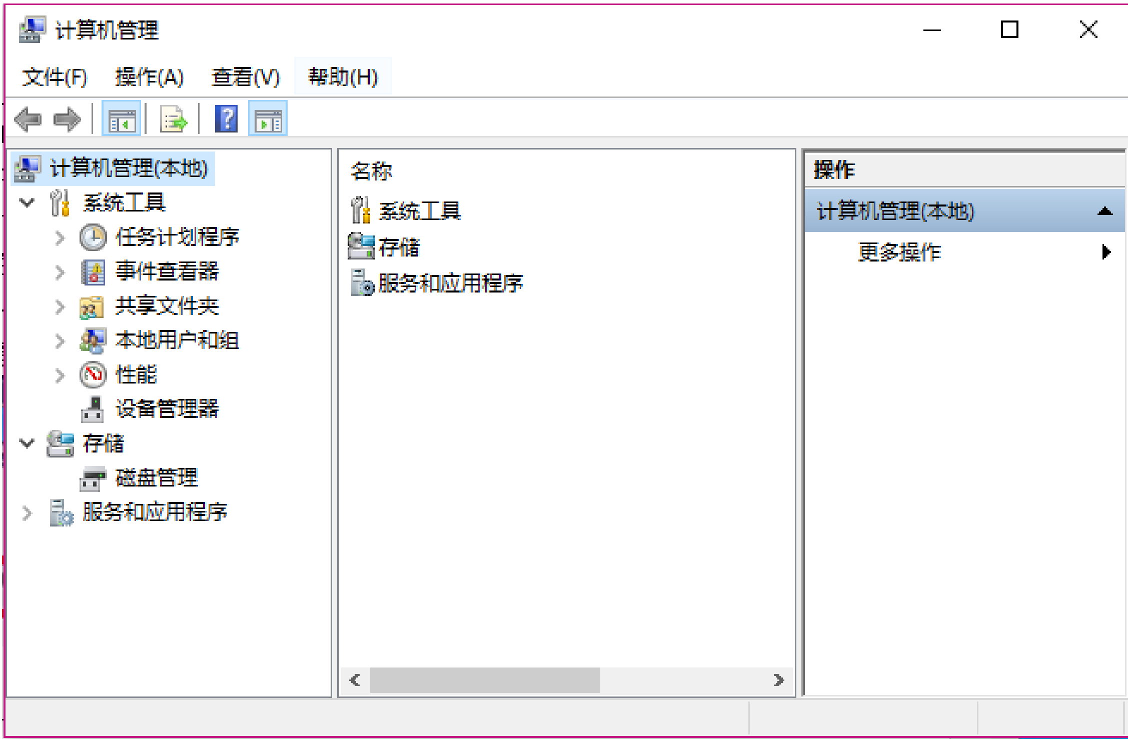


图 4.6.5 计算机管理器

在图 4.6.5 中，点击左侧“计算机管理(本地)”中的“设备管理器”，在右侧选中“端口(COM 和 LPT)”，如图 4.6.6 所示：

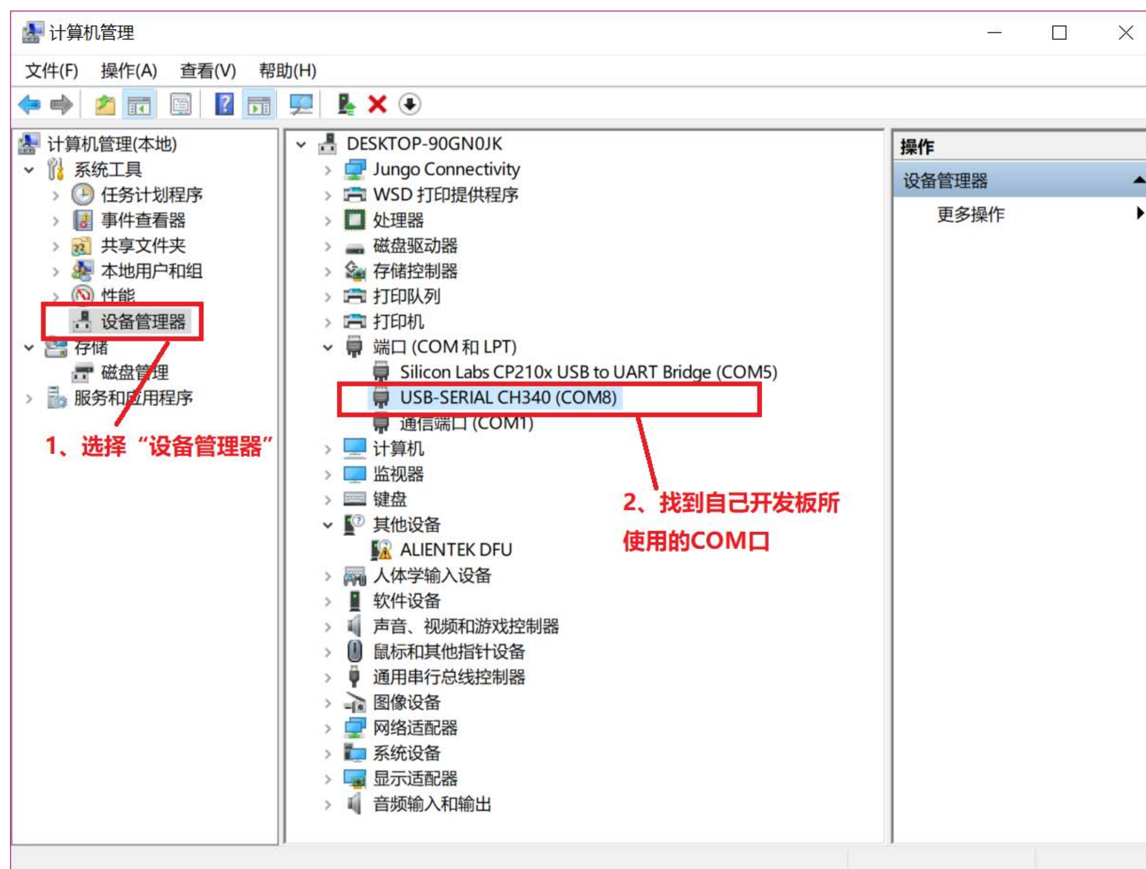


图 4.6.6 设备管理器

如果在图 4.6.6 中找到了有“USB-SERIAL CH340”字样的端口设备就说明 CH340 驱动成功了，一定要用 USB 线将开发板的串口和电脑连接起来！！！！

4.7 SecureCRT 软件安装和使用

4.7.1 SecureCRT 安装

在后续的开发过程中我们需要在 Windows 下使用 SecureCRT 作为终端，SecureCRT 支持 SSH 以及串口，我们通常使用 SecureCRT 来作为串口终端使用。SecureCRT 下载地址为：https://www.vandyke.com/download/index.html,下载界面如图 4.7.1 所示：

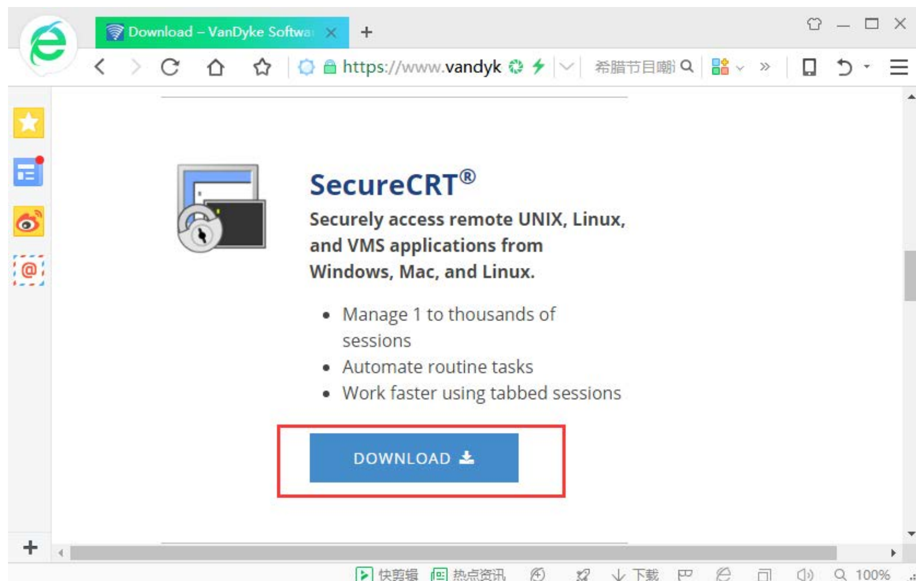


图 4.7.1 SecureCRT 下载界面

我们已经下载好放到开发板光盘中了，路径为：开发板光盘->3、软件->SecureCRT7.1，我们提供了两个版本的软件: scrt712-x86.exe 和 scrt733-x64.exe，这两个分别为 32 位和 64 位，大家根据自己所使用的电脑来选择安装版本，我的电脑是 64 位的，因此安装 scrt733-x64.exe。双击 scrt733-x64.exe 开始安装，界面如图 4.7.1.1 所示：

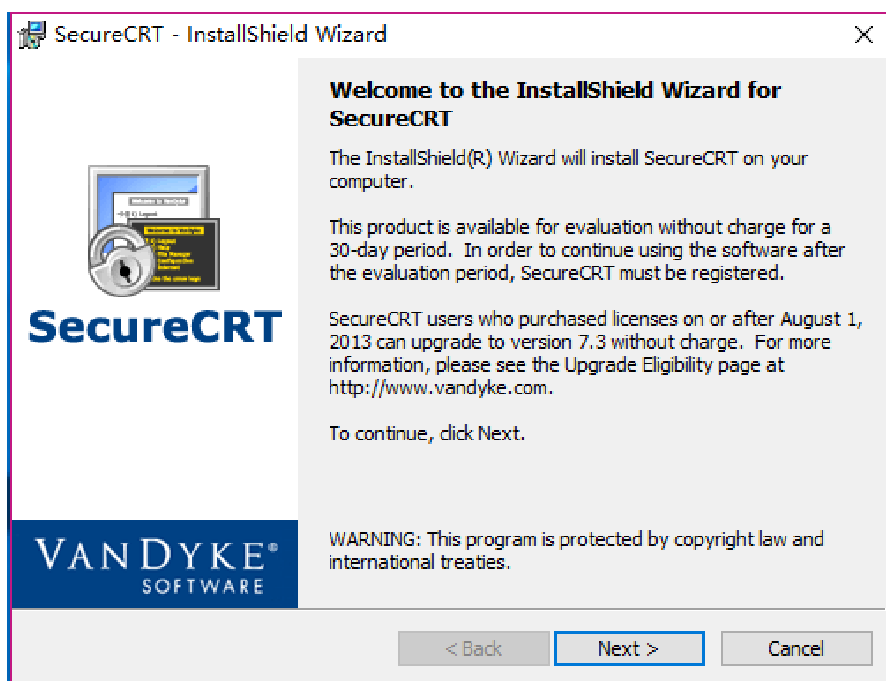


图 4.7.1.1 安装欢迎界面

点击图 4.7.1.1 中的“Next”按钮，进入 License 许可界面，如图 4.7.1.2 所示：

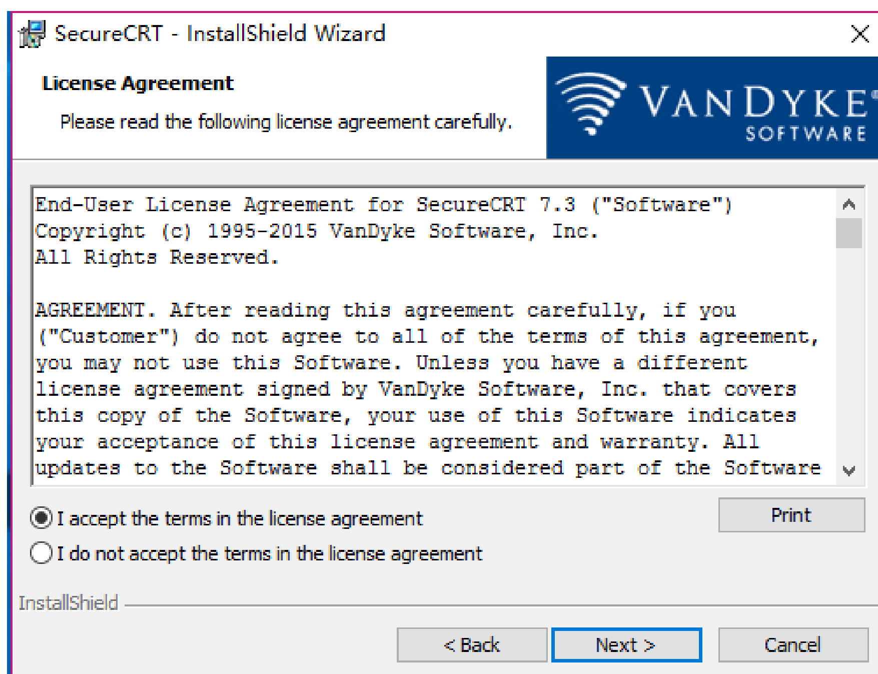


图 4.7.1.2 License 界面

在图 4.7.1.2 中，选择“I accept the terms in the license agreement”，然后点击“Next”按钮，进入使用者选择界面，如图 4.7.1.3 所示：

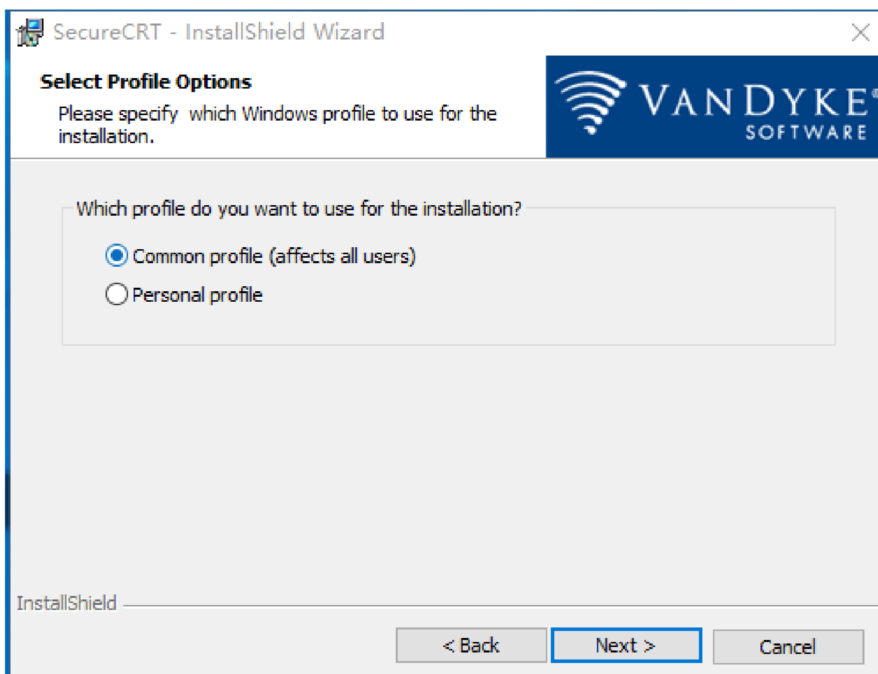


图 4.7.1.3 使用者选择

选择“Common profile(affects all users)”，也就是所有登陆到此电脑的用户都可以使用 SecureCRT，选中以后点击“Next”进入下一步，进入安装类型选择界面，如图 4.7.1.4 所示：

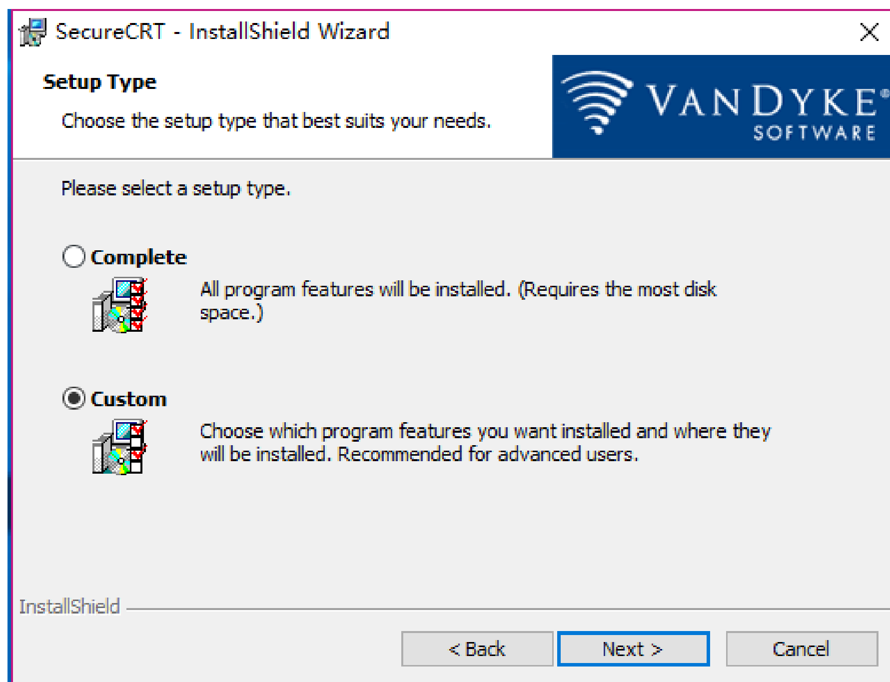


图 4.7.1.4 安装类型

在图 4.7.1.4 中我们选择“Custom”，也就是自定义安装，自定义安装我们可以选择安装目录，选择好以后点击“Next”进入下一步。进入安装路径选择界面，如图 4.7.1.5 所示

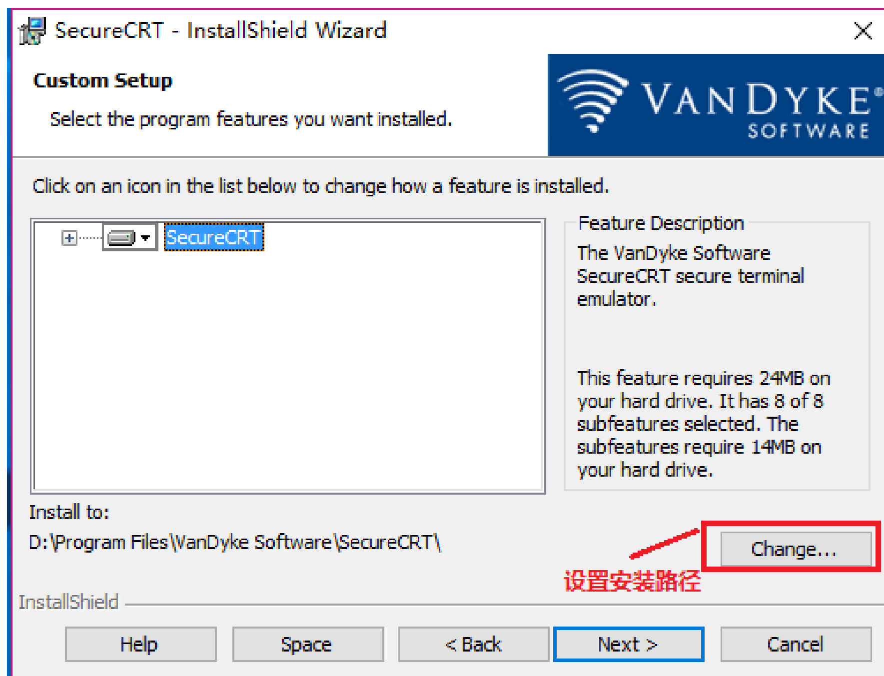


图 4.7.1.5 安装路径选择

根据自己实际情况设置 SecureCRT 安装路径，设置好以后点击“Next”按钮，下一个界面让你选择是否在桌面创建图标，默认是需要创建的，所以我们不用做任何修改，直接点击“Next”，进入图 4.7.1.6 所示界面：

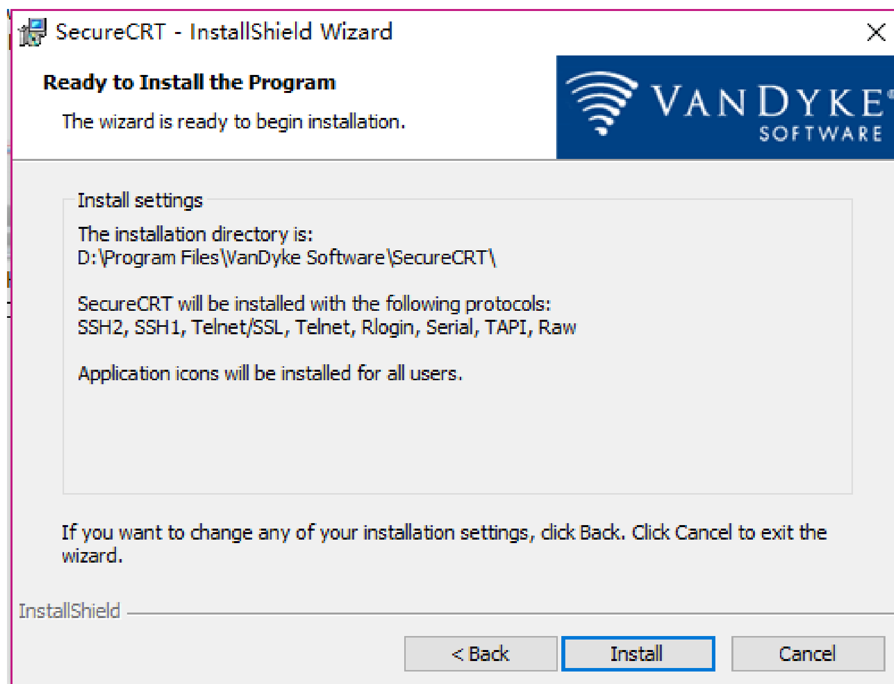


图 4.7.1.6 安装确认

点击图 4.7.1.6 中的“Install”按钮正式开始安装，等待安装完成界面，安装完成以后如图 4.7.1.7 所示：

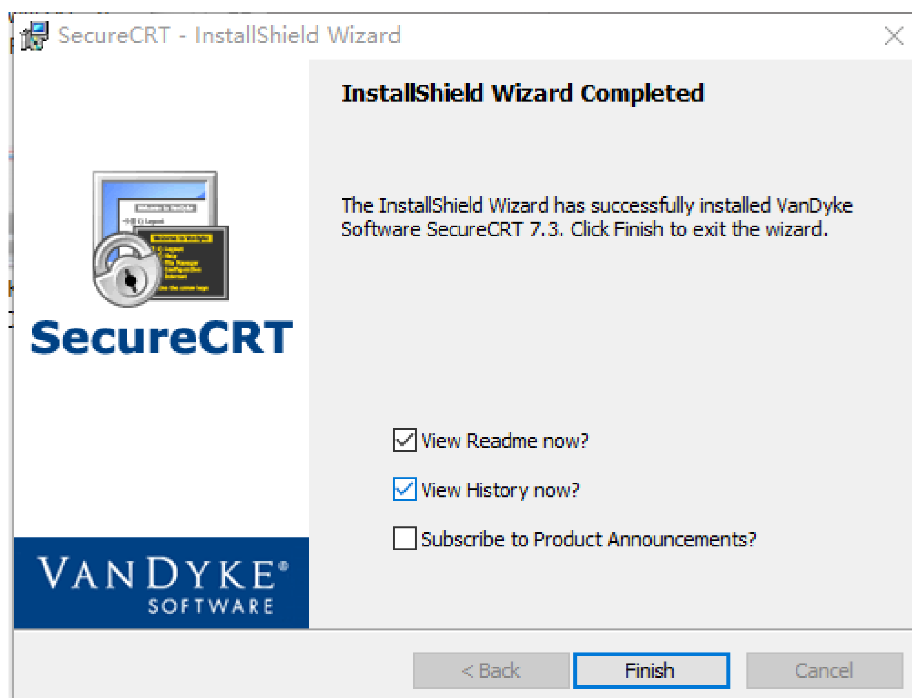


图 4.7.1.7 安装完成

点击图 4.7.1.7 中的“Finish”按钮退出安装，至此 SecureCRT 安装成功，安装成功以后就会在桌面出现相应的图标，如图 4.7.1.8 所示：

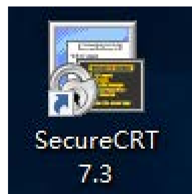


图 4.7.1.8 SecureCRT 图标

4.7.2 SecureCRT 使用

SecureCRT 功能很强大，支持 SSH，可以用来远程登陆；支持串口，可以用来作为 Linux 开发板的串口终端。我们用的最多的就是将 SecureCRT 作为串口终端来使用。双击图 4.6.1.8 所示的 SecureCRT 图标，打开 SecureCRT，第一次打开界面如图 4.6.2.1 所示的无效许可对话框：

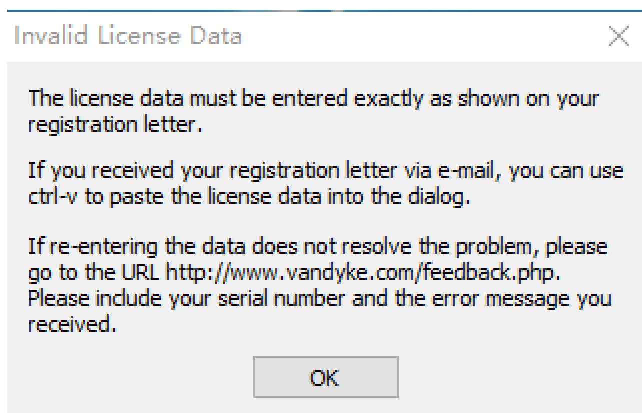


图 4.7.2.1 无效许可

因为 SecureCRT 也是付费软件，所以会弹出无效许可对话框，点击“OK”按钮，弹出序列号输入对话框，如图 4.7.2.2 所示：

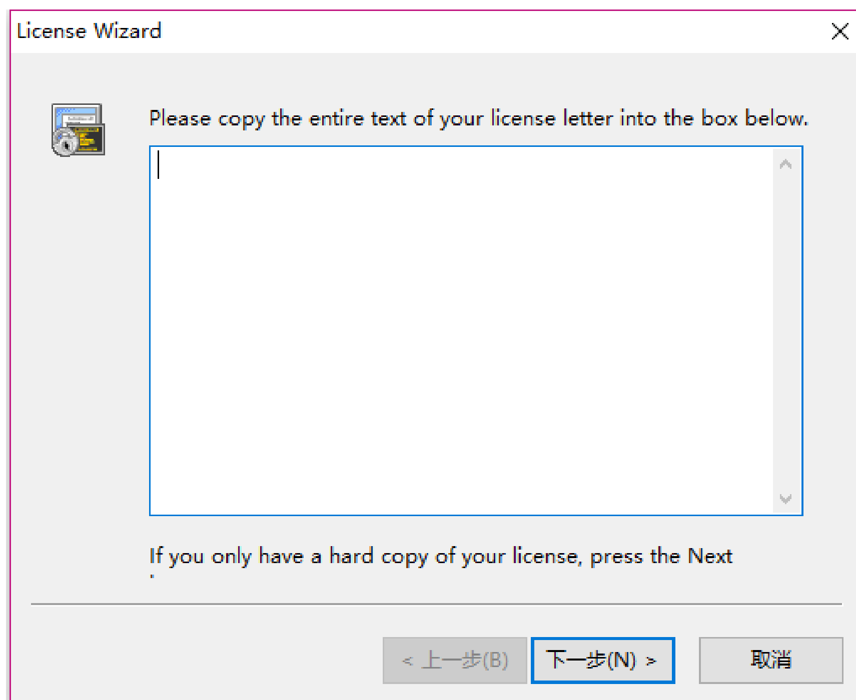


图 4.7.2.2 序列号输入

如果购买了序列号的话就可以输入序列号进行注册，注册成功以后就会进入到 SecureCRT

主界面，如图 4.6.2.3 所示：

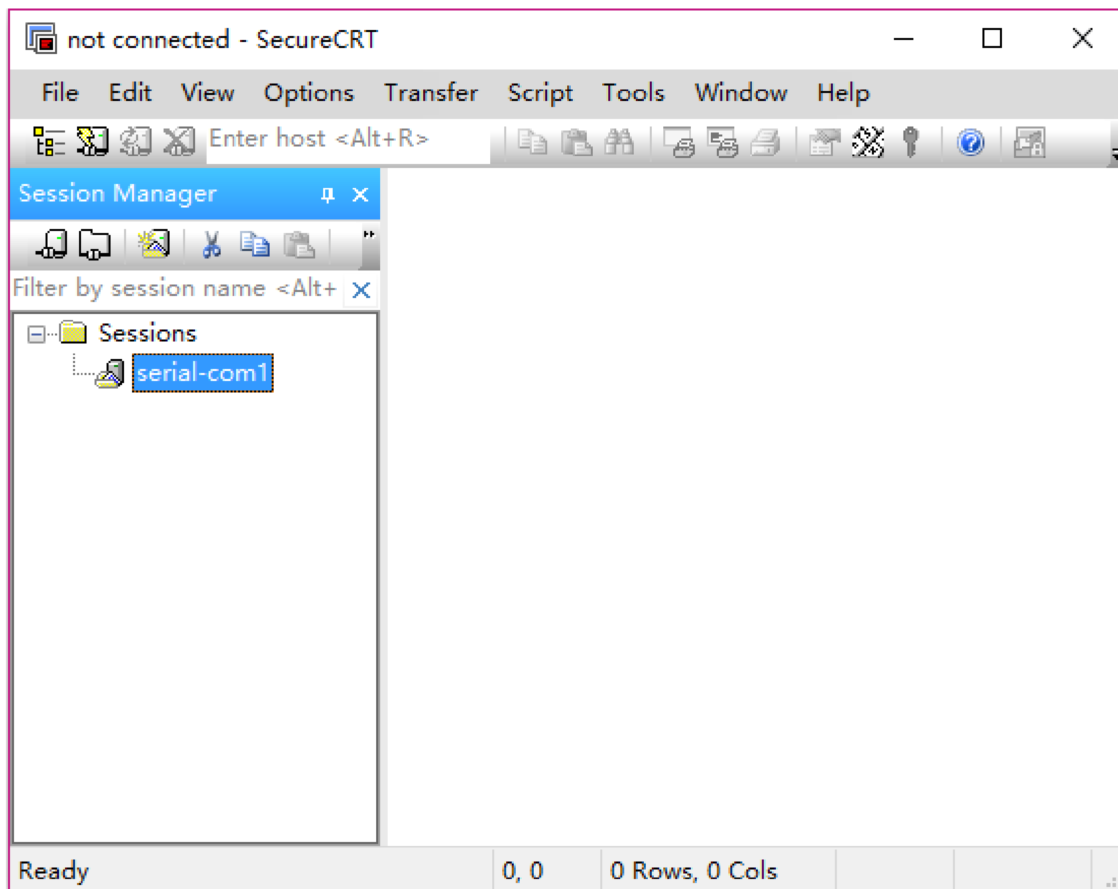


图 4.7.2.3 SecureCRT 主界面

我们以串口连接为例讲解如何使用 SecureCRT，我们需要准备好一个能进行串口通信的设备，我们的 I.MX6U-ALPHA 开发板就可以。I.MX6U-ALPHA 开发板出厂已经烧写了 Linux 系统，Linux 系统在运行的过程中会通过串口输出信息，通过串口可以实现 Linux 命令行交互操作，就和 Ubuntu 里面的终端一样，使用方法如下：

1、查看开发板当前使用的串口号

首先通过 USB 线将开发板的串口和电脑连接起来，打开“设备管理器”，在设备管理器中查看当前连接到电脑的端口都有哪些，如图 4.7.2.4 所示：

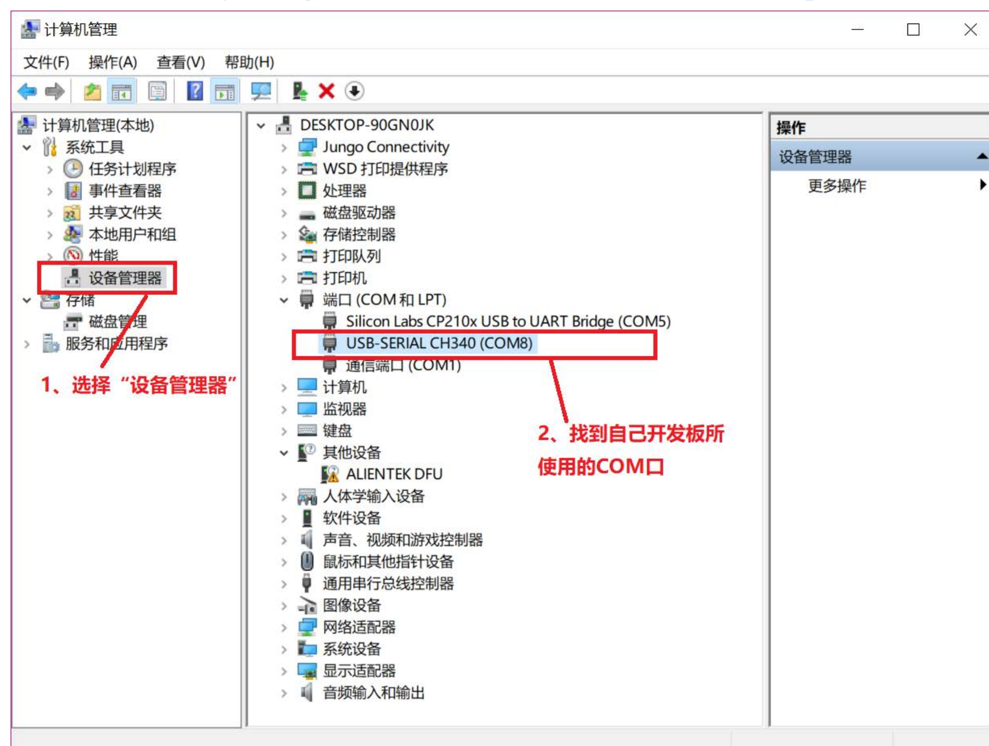


图 4.7.2.4 设备管理器

在图 4.7.2.4 中可以看到有多个 COM 口，哪个才是我们开发板的呢？I.MX6U-ALPHA 开发板使用的 CH340 芯片完成串口转 USB，所以“USB-SERIAL CH340(COM8)”就是我的开发板所使用的端口，串口号为 COM8。如果你的电脑连接了多个 CH340 做的 USB 转串口设备，无法区分哪个才是开发板所使用的，只需要把你的开发板串口拔掉，看看哪个串口号消失了，然后再重新插上开发板的串口线，再看一下那个消失的串口号会不会重新出现，如果会的话那你的开发板就是用的这个串口号。

2、设置 SecureCRT

我们已经知道了当前开发板所使用的串口号了，比如我的是 COM8，打开 SecureCRT，然后点击 File->Quick Connect…，如图 4.7.2.5 所示：

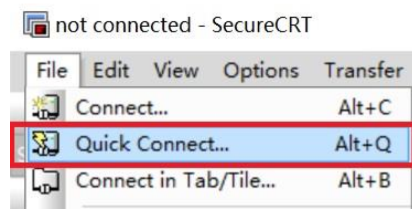


图 4.7.2.5 打开快速连接

打开以后的快速连接界面如图 4.7.2.6 所示：

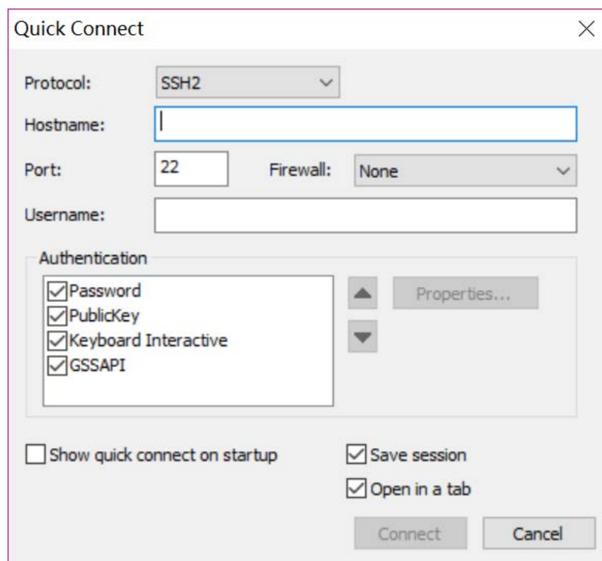


图 4.7.2.6 快速连接

按照图 4.7.2.7 所示进行设置：

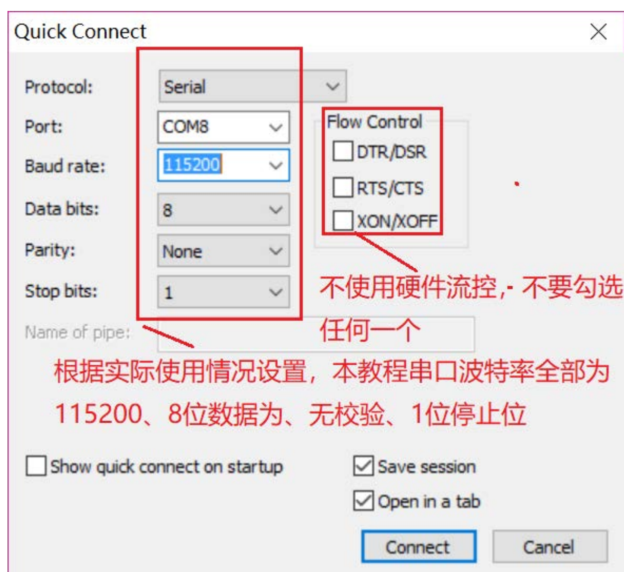


图 4.7.2.7 串口设置

设置好以后点击“Connect”按钮进行连接，连接成功以后 SecureCRT 如图 4.7.2.8 所示：

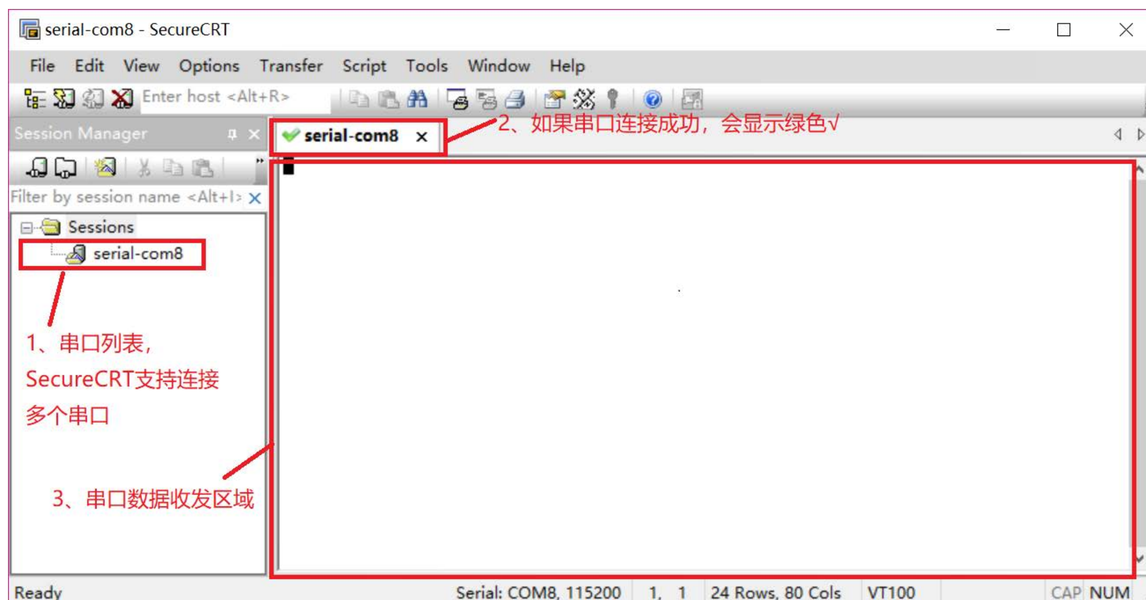


图 4.7.2.8 串口连接成功界面

在图 4.7.2.8 中，左侧是会话列表，保存着历史会话，会显示出所有曾经连接的串口，这个在关闭 SecureCRT 以后会被保存起来，下次重新打开 SecureCRT 就可以直接使用这个串口会话连接进行快速连接。比如我们关闭 SecureCRT，在关闭 SecureCRT 之前要先关闭所有的会话(串口)，重新打开 SecureCRT，如图 4.7.2.9 所示：

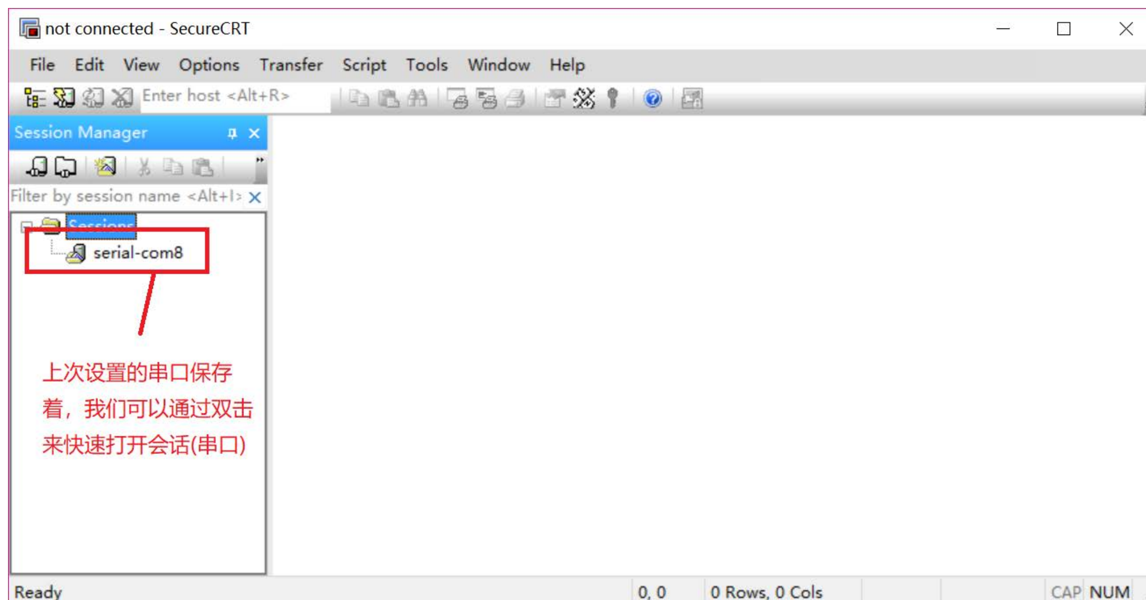


图 4.7.2.9 重新打开 SecureCRT

图 4.7.2.9 中重新打开的 SecureCRT 保存这上次关闭之前建立的会话(串口)“serial-com8”，通过双击“serial-com8”可以重新连接会话(串口)，不需要再使用快速连接对话框进行连接设置。

I.MX6U-ALPHA 开发板默认出厂烧写了 Linux 系统，所以如果连接上 SecureCRT 以后会将串口作为终端，会输出 Linux 系统启动信息，并且可以通过 SecureCRT 来操作开发板中的 Linux 系统，此时 SecureCRT 就是开发板的终端，和 Ubuntu 中的终端一样，如图 4.7.2.10 所示：

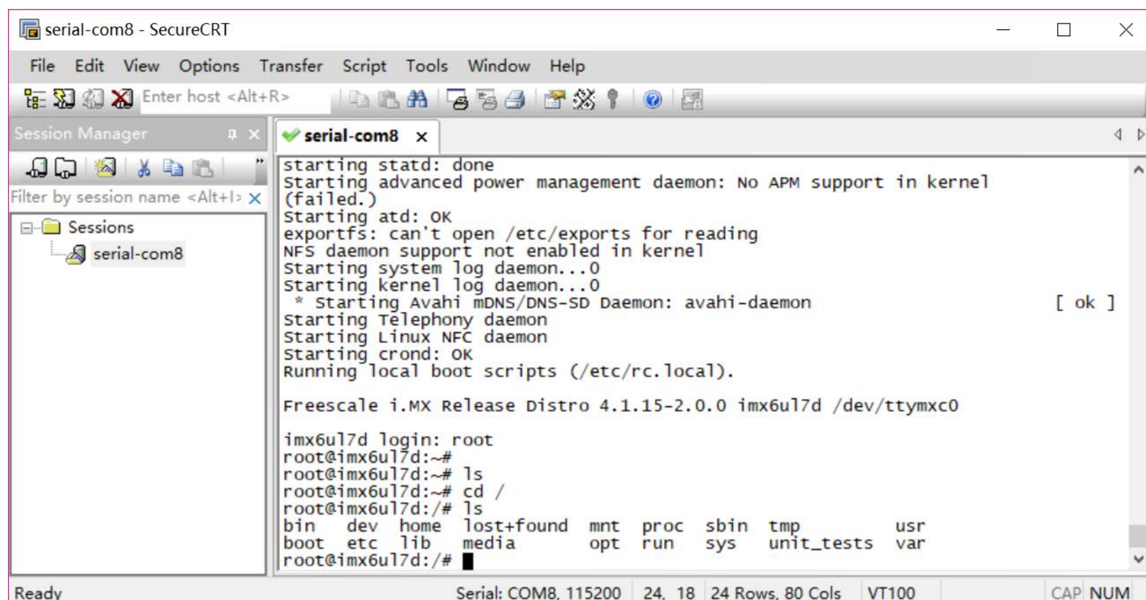


图 4.7.2.10 SecureCRT 作为 Linux 终端

4.8 Putty 软件的安装和使用

4.8.1 Putty 软件安装

Putty 和 SecureCRT 是类似的软件，都是用来作为 SSH 或者串口终端的，区别在于 SecureCRT 是付费软件，而 Putty 是免费的！！！这点很重要啊！虽然 Putty 没有 SecureCRT 功能强大，但是 Putty 用来作为嵌入式 Linux 的串口终端是绰绰有余。Putty 在官网下载即可，下载地址为：https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html，下载界面如图 4.8.1.1 所示：

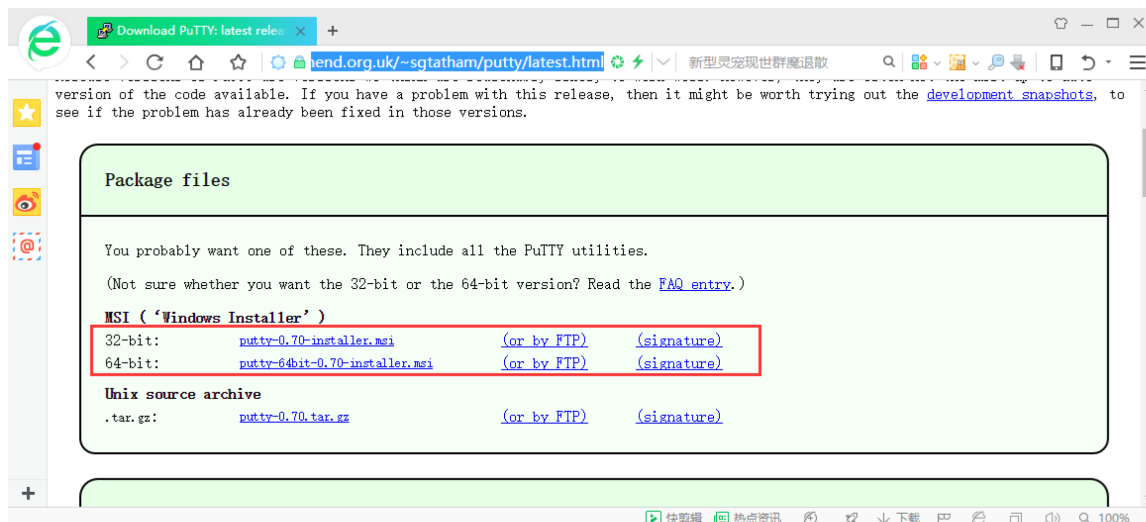


图 4.8.1.1 Putty 下载界面

Putty 同样提供了 32 位和 64 位两个版本的软件，我们已经下载好放到开发板光盘中了，路径为：开发板光盘->3、软件->Putty，有 32 位和 64 位两种，putty-0.70-installer.msi 是 32 位版本的，putty-64bit-0.70-installer.msi 为 64 位版本的，根据自己所使用的 Windows 系统选择合适的版本。因为我的电脑是 64 位系统，所以我使用的是 putty-64bit-0.70-installer.msi，双击开始安装，安装界面如图 4.8.1.2 所示：

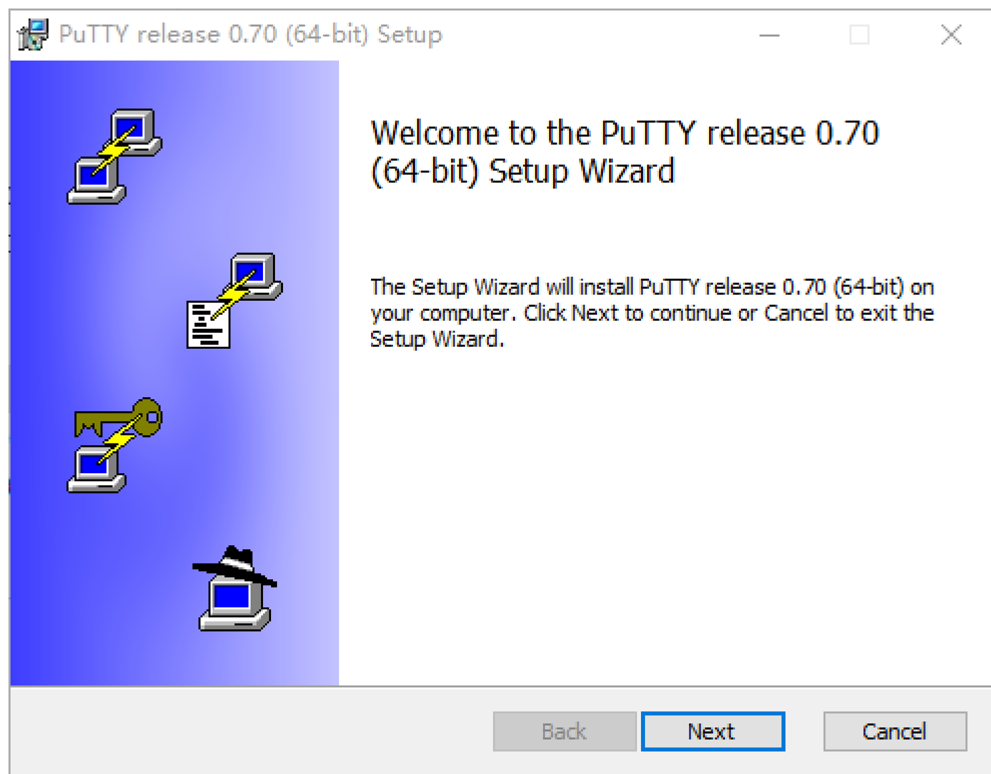


图 4.8.1.2 Putty 安装界面

点击图 4.7.1.2 中的“Next”按钮，进入下一步，下一步是选择安装路径，大家根据自己的实际情况选择一个安装路径，如图 4.8.1.3 所示：

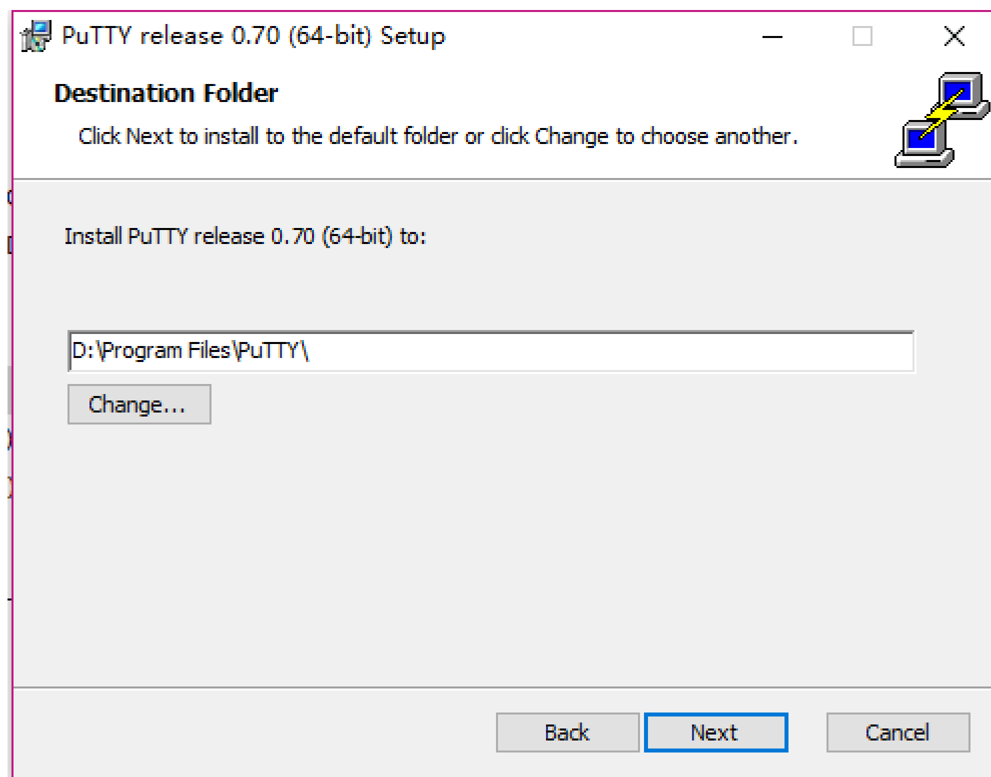


图 4.8.1.3 安装路径

设置好安装路径以后点击“Next”按钮进入下一步，如图 4.8.1.4 所示：

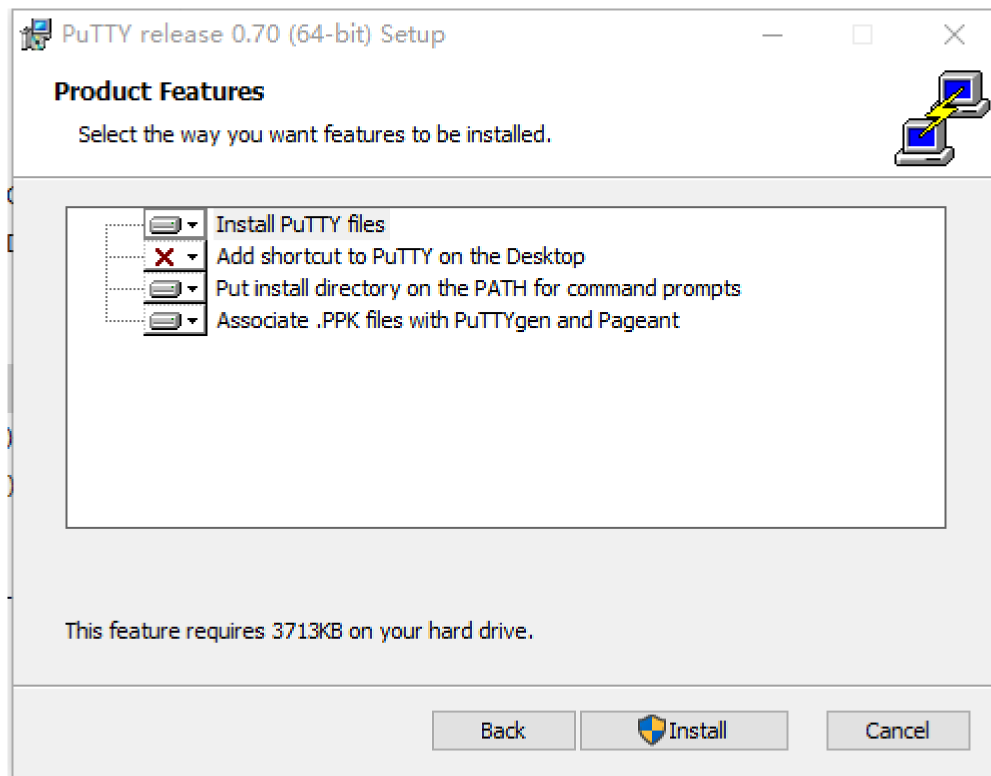


图 4.8.1.3 产品特性

点击图 4.8.1.3 中的“Install”按钮，开始安装，安装完成以后如图 4.8.1.4 所示：

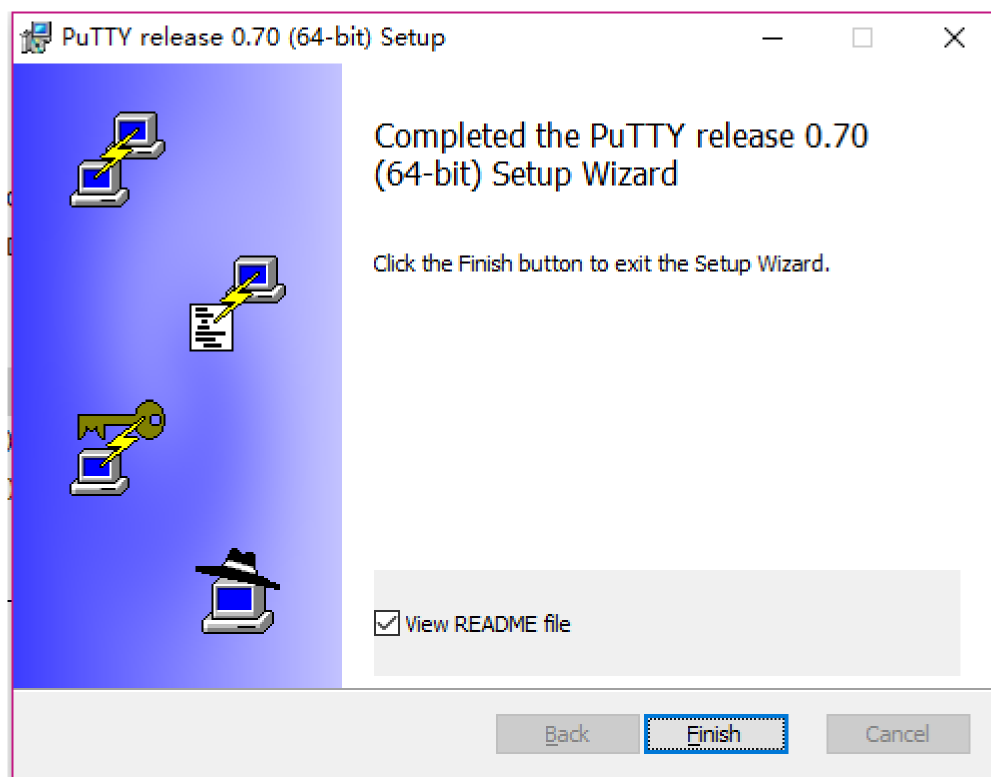


图 4.8.1.4 安装完成

点击图 4.8.1.4 中的“Finish”按钮退出安装。Putty 安装完成以后桌面可能不会出现 APP 图标，自行找到安装目录，将 Putty 图标的快捷方式发送到桌面上即可，Putty 图标如图 4.8.1.5 所

示：



图 4.8.1.5 Putty 图标

4.8.2 Putty 软件使用

使用 USB 线将开发板串口和电脑连接起来，打开 Putty 软件，打开以后是配置界面，如图 4.8.2.1 所示：

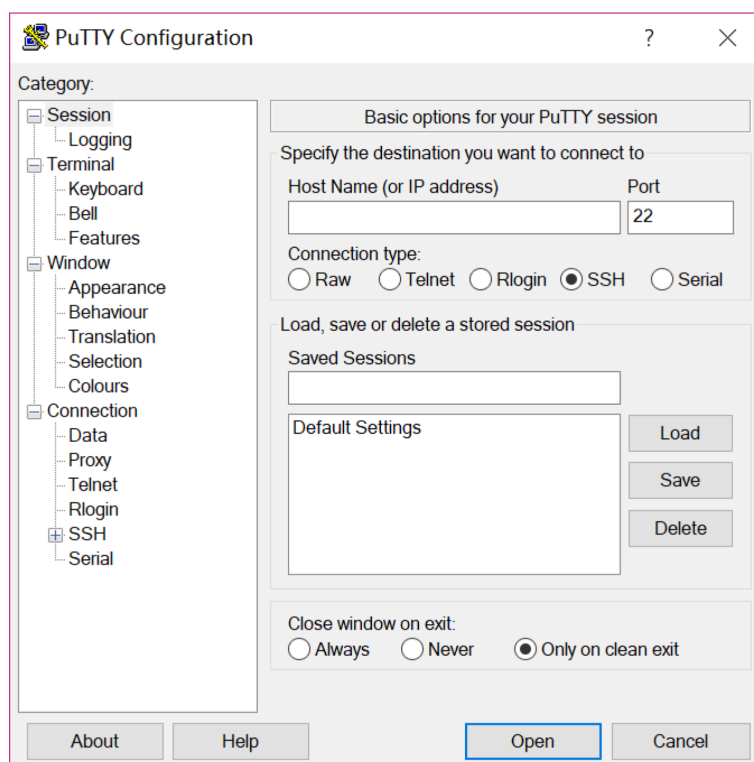


图 4.8.2.1 配置界面

我们要用到串口功能，所以在左侧选择“Serial”，然后在右侧配置串口，配置完成以后如图 4.8.2.2 所示：

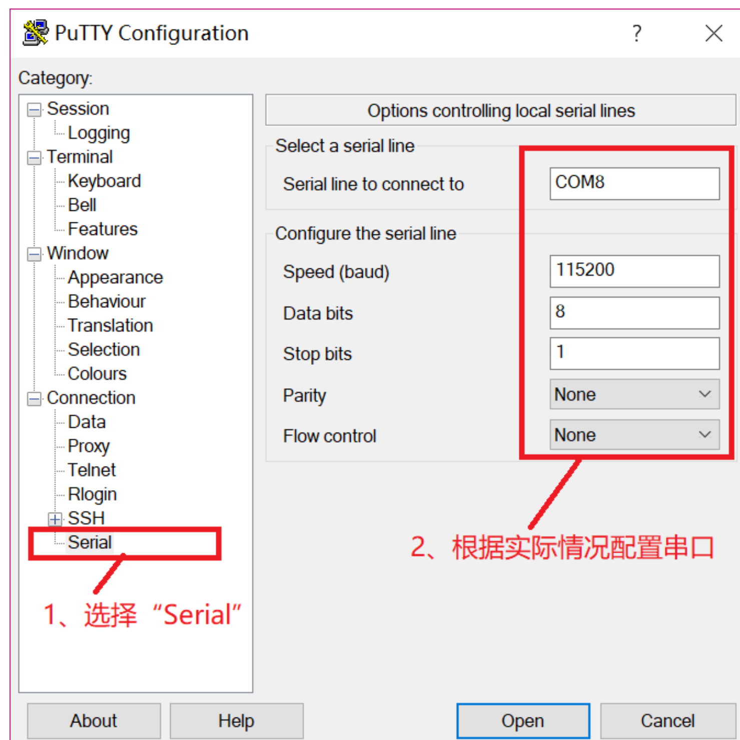


图 4.8.2.2 串口配置

按照图 4.8.2.2 配置好串口，配置好以后不要点击“Open”，没反应的！！我们还需要设置“Session”，设置如图 4.8.2.3 所示：

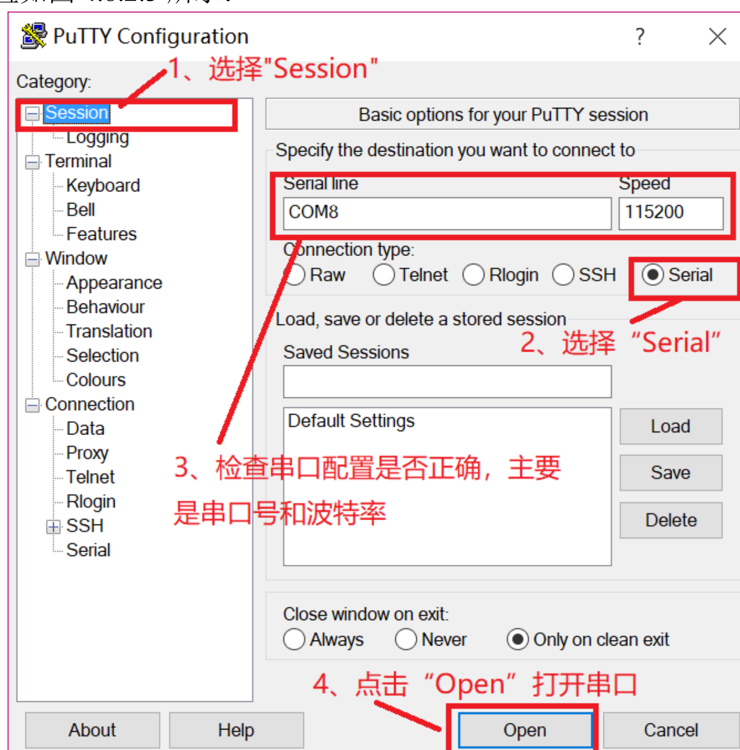


图 4.8.2.3 打开串口

按照图 4.8.2.3 设置好以后，点击“Open”打开串口，如果开发板里面烧写了 Linux 系统的话，Putty 就会显示 Linux 启动过程的信息，并且作为开发板的终端，如图 4.8.2.4 所示：

```
COM8 - PuTTY
U-Boot 2016.03-imx_v2016.03_4.1.15_2.0.0_ga+g0ec2a01 (Sep 16 2016 - 18:31:01 -05
00)

CPU:   Freescale i.MX6UL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 40C
Reset cause: POR
Board: MX6UL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

Display: TFT43AB (480x272)
Video: 480x272x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc1(part 0) is current device
Net:   FEC1
Error: FEC1 address not set.

Normal Boot
Hit any key to stop autoboot:  1
```

图 4.8.2.4 Putty 作为串口终端

相比于 SecureCRT 这种高富帅，Putty 就有点寒酸多了，但是 Putty 免费啊，至于要用哪一个大家自行选择一个合适的，本教程后面全部使用 SecureCRT。一是因为 SecureCRT 使用范围很广，几乎所有要用到串口终端的设备都使用 SecureCRT，二是 SecureCRT 功能强大。

第五章 I.MX6U-ALPHA 开发平台介绍

要学嵌入式 Linux 驱动开发肯定需要一个硬件平台，也就俗称的开发板，本书使用的是正点原子出品的 I.MX6U-ALPHA 开发板。这是一款以 NXP 的 I.MX6UL/ULL 为核心的 Cortex-A7 开发平台，板载资源丰富，非常适合以前学过 Cortex-M 内核单片机(比如 STM32)的工程进阶嵌入式 Linux 开发。工欲善其事，必先利其器，本章我们就先来详细了解一下未来将会陪伴我们很长一段时间的朋友——I.MX6U-ALPHA 开发平台。

5.1 正点原子 I.MX6U-ALPHA 开发板资源初探

正点原子目前已经拥有多款 STM32、I.MXRT 以及 FPGA 开发板，这些开发板常年稳居淘宝销量冠军，累计出货超过 10W 套。这款 ALPHA 开发板，是正点原子推出的第一款 Linux 开发板，采用底板+核心板的形式。接下来我们分别介绍 I.MX6U-ALPHA 开发板的底板和核心板。

5.1.1 I.MX6U-ALPHA 开发板底板资源

首先，我们来一下 I.MX6U-ALPHA 开发板的底板资源图，如图 5.1.1.1 所示：

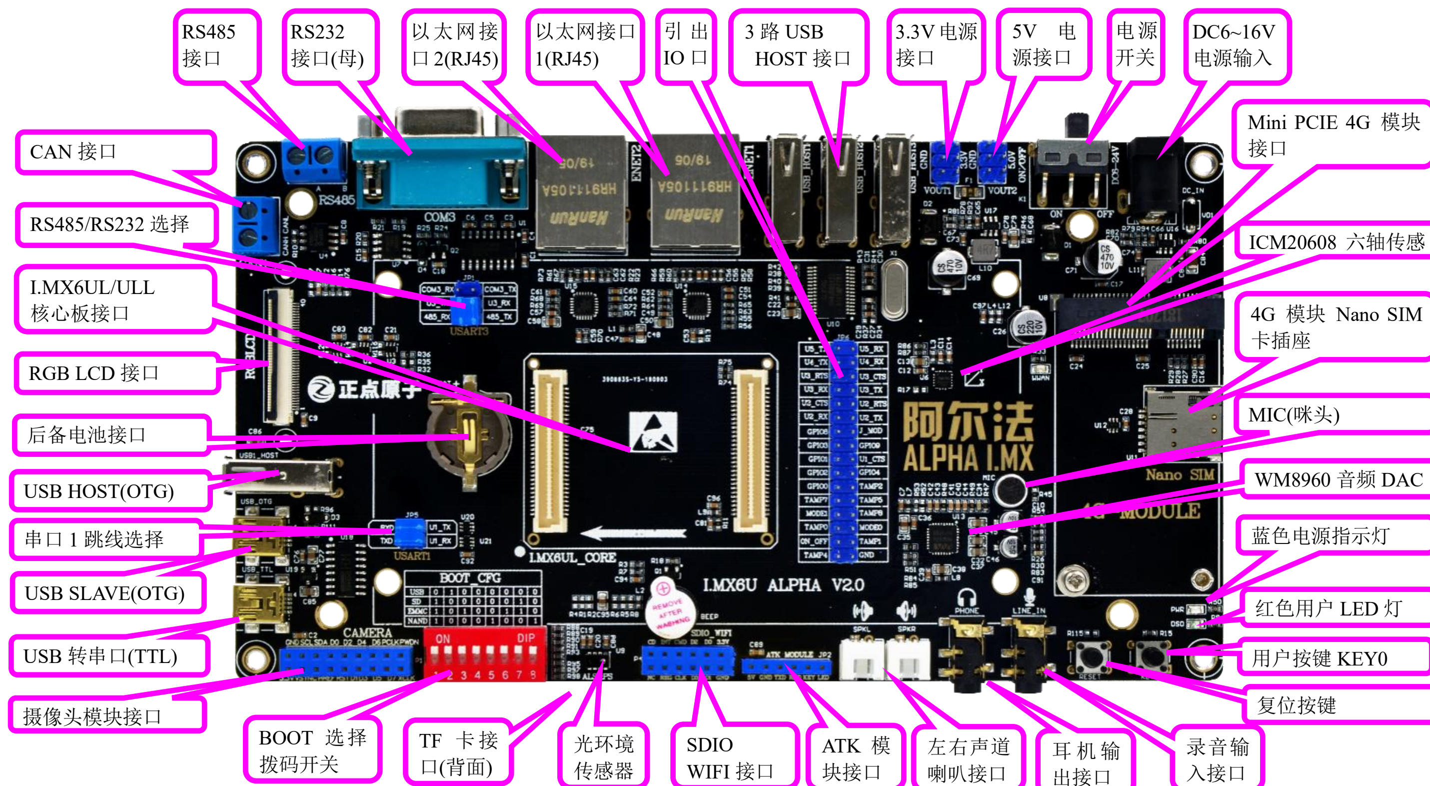


图 5.1.1.1 I.MX6U-ALPHA 开发板底板资源图

从图 5.1.1.1 可以看出，I.MX6U-ALPHA 开发板底板资源十分丰富，把 I.MX6UL/ULL 的内部资源发挥到了极致，基本所有 I.MX6UL/ULL 的内部资源都可以在此开发板上验证，同时扩充丰富的接口和功能模块，整个开发板显得十分大气。

开发板的外形尺寸为 100mm*180mm 大小，板子的设计充分考虑了人性化设计，并结合正点原子多年的开发板设计经验，经过多次改进，最终确定了这样的设计。

正点原子 I.MX6U-ALPHA 开发板底板板载资源如下：

- ◆ 1 个核心板接口，支持 I.MX6UL/6ULL 等核心板
- ◆ 1 个电源指示灯（蓝色）
- ◆ 1 个状态指示灯（红色）
- ◆ 1 个六轴（陀螺仪+加速度）传感器芯片，ICM20608
- ◆ 1 个高性能音频编解码芯片，WM8960
- ◆ 1 路 CAN 接口，采用 TJA1050 芯片
- ◆ 1 路 485 接口，采用 SP3485 芯片
- ◆ 1 路 RS232 串口（母）接口，采用 SP3232 芯片
- ◆ 1 个 ATK 模块接口，支持正点原子蓝牙/GPS/MPU6050/手势识别等模块

- 1 个光环境传感器（光照、距离、红外三合一）
- 1 个摄像头模块接口
- 1 个 OLED 模块接口
- 1 个 USB 串口，可用于代码调试
- 1 个 USB SLAVE(OTG)接口，用于 USB 从机通信
- 1 个 USB HOST(OTG)接口，用于 USB 主机通信
- 1 个有源蜂鸣器
- 1 个 RS232/RS485 选择接口
- 1 个串口选择接口
- 1 个 TF 卡接口（在板子背面）
- 2 个 10M/100M 以太网接口（RJ45）
- 1 个录音头（MIC/咪头）
- 1 路立体声音频输出接口
- 1 路立体声录音输入接口
- 1 个小扬声器（在板子背面）
- 2 个扬声器外接接口，左右声道。
- 1 组 5V 电源供应/接入口
- 1 组 3.3V 电源供应/接入口
- 1 个直流电源输入接口（输入电压范围：DC6~24V）
- 1 个启动模式选择配置接口
- 1 个 RTC 后备电池座，并带电池
- 1 个复位按钮，可用于复位 MPU 和 LCD
- 1 个功能按钮
- 1 个电源开关，控制整个板的电源
- 1 个 Mini PCIE 4G 模块接口
- 1 个 Nano SIM 卡接口
- 1 个 SDIO WIFI 接口

正点原子 I.MX6U-ALPHA 开发板底板的特点包括：

1） 接口丰富。板子提供十来种标准接口，可以方便的进行各种外设的实验和开发。
2） 设计灵活。我们采用核心板+转接板+底板形式，板上很多资源都可以灵活配置，以满足不同条件下的使用；我们引出了 105 个 IO 口，极大的方便大家扩展及使用。
3） 资源丰富。板载高性能音频编解码芯片、六轴传感器、百兆网卡、光环境传感器以及各种接口芯片，满足各种应用需求。
4） 人性化设计。各个接口都有丝印标注，且用方框框出，使用起来一目了然；部分常用外设大丝印标出，方便查找；接口位置设计合理，方便顺手。资源搭配合理，物尽其用。

5.1.2 I.MX6U 核心板资源

接下来，我们来看 I.MX6ULL 核心板资源图，正点原子的 I.MX6ULL 核心板根据存储芯片的不同分为 EMMC 和 NAND 两种，根据对外提供的接口可以分为邮票孔和 BTB 两种。本教程主要使用与教学的，所以只会讲解 BTB 接口的核心板，BTB 接口的 NAND 版本核心板如图 5.1.2.1 所示：

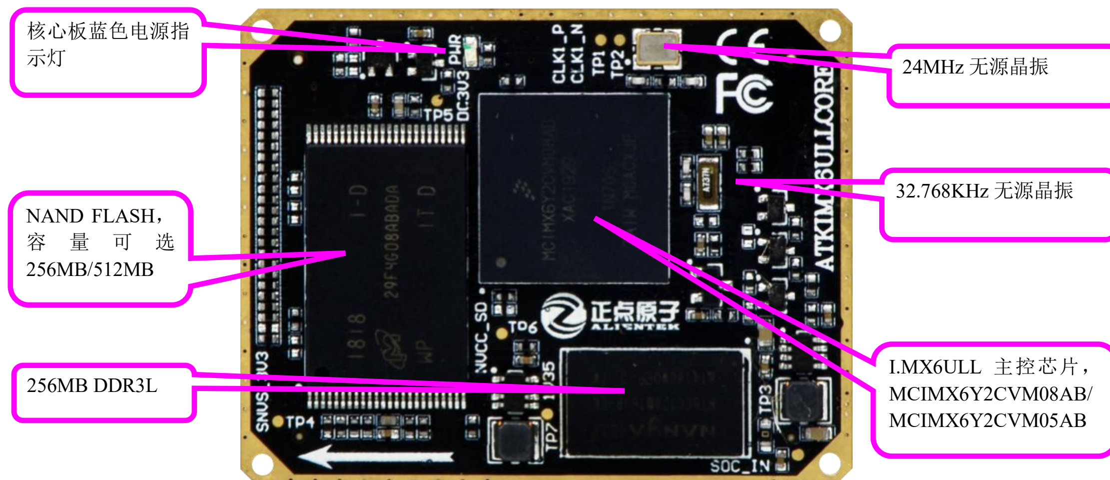


图 5.1.2.1 I.MX6ULL NAND BTB 接口核心板资源图

从图 5.1.2.1 可以看出，I.MX6ULL 核心板板载资源丰富，可以满足各种应用的需求。整个核心板的外形尺寸为 46mm*36mm 大小，非常小巧，并且采用了贴片板对板连接器，使得其可以很方便的应用在各种项目上。I.MX6ULL NAND 版核心板为工业级工作温度，可以应用在温度要求严格的场合。

正点原子 I.MX6ULL NAND 版核心板板载资源如下：

- CPU：MCIMX6Y2CVM05AB(工业级)或 MCIMX6Y2CVM08AB（工业级），主频分别为 528MHz 和 800MHz(实际为 792MHz)，BGA289
- 外扩 DDR3L：NT5CC128M16JR-EK，256MB 字节，工业级
- NAND FLASH：MT29F2G08ABAEAWP-IT 或 MT29F4G08ABADAWP-IT，分别为 256MB/512MB 字节，均为工业级
- 两个 2*30 的防反插 BTB 座，共引出 120 PIN 。

BTB 接口的 EMMC 版本核心板如图 5.1.2.2 所示：

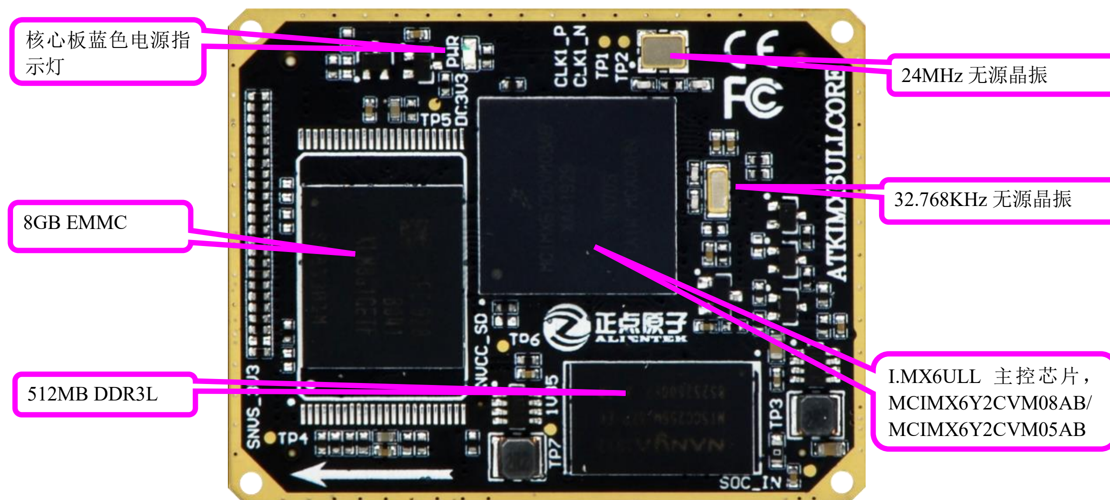


图 5.1.2.2 I.MX6ULL EMMC BTB 接口核心板资源图

从图 5.1.2.2 可以看出，EMMC 版本的核心和 NAND 版本的基本一样，不同之处在于将 NAND 换成了 EMMC，将 DDR3L 换成了 512MB 的商业级。因此正点原子的 EMMC 核心板为商业级的工作温度范围，如需工业级的请联系定制。

正点原子 I.MX6ULL EMMC 版核心板板载资源如下：

- CPU：MCIMX6Y2CVM08AB（工业级），800MHz(实际 792MHz)，BGA289

◆ 外扩 DDR3L：NT5CC256M16EP-EK，512MB 字节，商业级。
◆ EMMC：KLM8G1GET，这是一个 8GB 的 EMMC 芯片。
◆ 两个 2*30 的防反插 BTB 座，共引出 120 PIN

正点原子 I.MX6ULL 核心板的特点包括：

1） 体积小巧。核心板仅 46mm*36mm 大小，方便使用到各种项目里面。
2） 集成方便。核心板使用 120P BTB 连接座，可以非常方便的集成到客户 PCB 上，更换简单，方便维修测试。
3） 资源丰富。核心板板载：256MB/512MB DDR3L、可以选择 NAND 或 EMMC 等存储器，可以满足各种应用需求。
4） 性能稳定。核心板采用 6 层板设计，单独地层、电源层，且关键信号采用等长线走线，保证运行稳定、可靠。
5） 不管是 NAND 还是 EMMC 核心板均通过了 CE 和 FCC 认证。
6） 人性化设计。底部放有详细丝印，方便安装；按功能分区引出 IO 口，方便布线。

5.2 正点原子 I.MX6U-ALPHA 开发板资源说明

资源说明部分，我们将分为两个部分说明：硬件资源说明和软件资源说明。

5.2.1 硬件资源说明

这里我们首先详细介绍 I.MX6U-ALPHA 开发板的各个部分，包括底板和核心板二部分（图 5.1.1.1、图 5.1.2.1 和图 5.1.2.2 中的标注部分）的硬件资源，我们将按逆时针的顺序依次介绍。首先，我们来看底板的资源说明：

1. CAN 接口

这是开发板板载的 CAN 总线接口（CAN），通过 2 个端口和外部 CAN 总线连接，即 CANH 和 CANL。这里提醒大家：CAN 通信的时候，必须 CANH 接 CANH，CANL 接 CANL，否则可能通信不正常！

2. RS232/485 选择接口

这是开发板板载的 RS232（COM3）/485 选择接口（JP1），因为 RS485 基本上就是一个半双工的串口，为了节约 IO，我们把 RS232（COM3）和 RS485 共用一个串口，通过 JP1 来设置当前是使用 RS232（COM3）还是 RS485。这样的设计还有一个好处。就是我们的开发板既可以充当 RS232 到 TTL 串口的转换，又可以充当 RS485 到 TTL485 的转换。（注意，这里的 TTL 高电平是 3.3V）

3. I.MX6UL/ULL 核心板接口

这是开发板底板上面的核心板接口，由 2 个 2*30 的贴片板对板接线端子（3710F 公座）组成，可以用来插正点原子的 I.MX6UL/ULL 核心板等，从而学习 I.MX6UL/6ULL 等芯片，达到一个开发板，学习多款 SOC 的目的，减少重复投资。

4. RGBLCD 接口

这是转接板自带的 RGB LCD 接口（LCD），可以连接各种正点原子的 RGB LCD 屏模块，并且支持触摸屏（电阻/电容屏都可以）。采用的是 RGB888 格式，可显示 1677 万色，色彩显示丰富。

5. 后备电池接口

这是 I.MX6UL/ULL 后备区域的供电接口，可以用来给 I.MX6UL/ULL 的后备区域提供能量，在外部电源断电的时候，维持 SNVS 区域数据的存储，以及 RTC 的运行。

6. USB HOST(OTG)

这是开发板板载的一个侧插式的 USB-A 座（USB_HOST），由于 I.MX6U 的 USB 支持 OTG 功能，所以 USB 既可作 HOST，又可做 SLAVE。我们可以通过这个 USB-A 座，连接 U 盘/USB 鼠标/USB 键盘等其他 USB 从设备，从而实现 USB 主机功能。不过特别注意，由于 USB HOST 和 USB SLAVE 是共用一个 USB 端口，所以两者不可以同时使用。

7. USB 串口/串口 1

这是 USB 串口同 I.MX6U 的串口 1 进行连接的接口（JP5），标号 RXD 和 TXD 是 USB 转串口的 2 个数据口（对 CH340C 来说），而 U1_TX(TXD)和 U1_RX(RXD)则是 I.MX6U 串口 1 的两个数据口。他们通过跳线帽对接，就可以和连接在一起了，从而实现 I.MX6U 的串口通信。

设计成 USB 串口，是出于现在电脑上串口正在消失，尤其是笔记本，几乎清一色的没有串口。所以板载了 USB 串口可以方便大家调试。而在板子上并没有直接连接在一起，则是出于使用方便的考虑。这样设计，你可以把 I.MX6U-ALPHA 开发板当成一个 USB 转 TTL 串口，来和其他板子通信，而其他板子的串口，也可以方便地接到开发板上。

8. USB SLAVE(OTG)

这是开发板板载的一个 MiniUSB 头（USB_SLAVE），用于 USB 从机（SLAVE）通信，与上面的 USB HOST 一起作为 OTG 功能。通过此 MiniUSB 头，开发板就可以和电脑进行 USB 通信了。注意：该接口不能和 USB HOST 同时使用。

开发板总共板载了两个 MiniUSB 头，一个（USB_TTL）用于 USB 转串口，连接 CH340C 芯片；另外一个（USB_SLAVE）用于 I.MX6U 内部 USB。同时开发板可以通过此 MiniUSB 头供电，板载两个 MiniUSB 头（不共用），主要是考虑了使用的方便性，以及可以给板子提供更大的电流（两个 USB 都接上）这两个因素。

9. USB 转串口

这是开发板板载的另外一个 MiniUSB 头（USB_TTL），用于 USB 连接 CH340C 芯片，从而实现 USB 转串口。同时，此 MiniUSB 接头也是开发板的电源提供口。

10. 摄像头模块接口

这是开发板板载的一个摄像头模块接口（P1），摄像头模块（需自备），对准插入到此插槽中。

11. 启动(BOOT)拨码开关

I.MX6U 支持多种启动方式，比如 SD 卡、EMMC、NAND、QSPI FALSH 和 USB 等，要想从某一种设备启动就必须先设置好启动拨码开关。I.MX6U-ALPHA 开发板用了一个 8P 的拨码开关来选择启动方式，正点原子开发板支持从 SD 卡、EMCM、NAND 和 USB 这四种启动方式，这四种启动方式对应的拨码开关拨动方式已经写在了开发板上。大家在使用的时候根据自己的实际需求设置拨码开关即可。

12. TF 卡接口

这是开发板板载的一个标准 TF 卡接口（TF_CARD），该接口在开发板的背面，采用小型的 TF 卡接口，USDHC 方式驱动，有了这个 TF 卡接口，就可以满足海量数据存储的需求。

13. 光环境传感器

这是开发板板载的一个光环境三合一传感器（U9），它可以作为：环境光传感器、近距离（接近）传感器和红外传感器。通过该传感器，开发板可以感知周围环境光线的变化，接近距离等，从而可以实现类似手机的自动背光控制。

14. SDIO WIFI 接口

这是开发板上的一个 SDIO WIFI(P4)接口，可以通过此接口连接正点原子出品的 SDIO WIFI 模块。SDIO WIFI 接口和 TF 卡共用一个 USDHC 接口，因此不能同时和 TF 卡使用。

15. ATK 模块接口

这是开发板板载的一个正点原子通用模块接口（JP2），目前可以支持正点原子开发的 GPS 模块、蓝牙模块、MPU6050 模块、激光测距模块和手势识别模块等，直接插上对应的模块，就可以进行开发。后续我们将开发更多兼容该接口的其他模块，实现更强大的扩展性能。

16. 左右声道喇叭接口

开发板板载了一个高性能的音频解码芯片 WM8960，此芯片可以驱动左右声道 2 个 8Ω，1W 的小喇叭，这两个接口用于外接两个左右声道小喇叭。不过在 I.MX6U-ALPHA 开发板的背面已经默认焊接了一个小喇叭，这个小喇叭接到了右声道上，因此如果要在此接口的右声道上外接小喇叭，那么必须先将开发板上自带的喇叭拆掉，否则 WM8960 驱动能力可能不足。

17. 耳机输出接口

这是开发板板载的音频输出接口（PHONE），该接口可以插 3.5mm 的耳机，当 WM8960 放音的时候，就可以通过在该接口插入耳机，欣赏音乐。此接口支持耳机插入检测，如果耳机不插入的话默认通过喇叭播放音乐，如果插入耳机的话就关闭喇叭，通过耳机播放音乐。

18. 录音输入接口

这是开发板板载的外部录音输入接口（LINE_IN）,通过咪头我们只能实现单声道的录音，而通过这个 LINE_IN，我们可以实现立体声录音。

19. 复位按键

这是开发板板载的复位按键（RESET），用于复位 I.MX6U，还具有复位液晶的功能，因为液晶模块的复位引脚和 I.MX6U 的复位引脚是连接在一起的，当按下该键的时候，I.MX6U 和液晶一并被复位。

20. 用户按键 KEY

这是开发板板载的 1 个机械式输入按键（KEY0），可以做为普通按键输入使用。

21. 红色用户 LED 灯

这是开发板板载的 1 个 LED 灯，为红色，用户可以使用此 LED 灯。在调试代码的时候，使用 LED 来指示程序状态，这是非常不错的一个辅助调试方法。

22. 蓝色电源指示 LED 灯

这是开发板电源指示 LED 灯，为蓝色，当板子供电正常的时候此灯就会常亮。如果此灯不亮的话就说明开发板供电有问题(排除 LED 灯本身损坏的情况)。

23. WM8960 音频 DAC

这是一颗欧胜公司出品的音频 DAC 芯片，用于实现音乐播放与录音。

24. MIC(咪头)

这是开发板的板载录音输入口（MIC），该咪头直接接到 WM8960 的输入上，可以用来实现录音功能。

25. Nano SIM 卡接口

这是开发板上的 Nano SIM 卡接口，如果要使用 4G 模块的话就需要在此接口中插入 Nano SIM 卡。

26. ICM20608 六轴传感器

这是开发板板载的一个六轴传感器芯片(U6)，型号为 ICM20608，此芯片采用 SPI 接口与 I.MX6U 相连接。ICM20608 内部集成 1 个三轴加速度传感器和 1 个三轴陀螺仪，该传感器在姿态测量方面应用非常广泛。所以喜欢玩姿态测量的朋友，也可通过本开发板进行学习。

27. Mini PCIE 4G 接口

这是开发板板载的一个 Mini PCIE 座，但是本质上走的 USB 协议，通过此接口可以连接 4G 模块，比如高新兴物联的 ME3630。接上 4G 模块以后 I.MX6U-ALPHA 开发板就可以实现 4G 上网功能，对于不方便布网线或者没有 WIFI 的场合来说是个不错的选择。

28. DC6~16V 电源输入

这是开发板板载的一个外部电源输入口（DC_IN），采用标准的直流电源插座。开发板板载了 DC-DC 芯片（JW5060T），用于给开发板提供高效、稳定的 5V 电源。由于采用了 DC-DC 芯片，所以开发板的供电范围十分宽，大家可以很方便的找到合适的电源（只要输出范围在 DC6~16V 的基本都可以）来给开发板供电。在耗电比较大的情况下，比如用到 4.3 屏/7 寸屏/网口的时候，建议使用外部电源供电，可以提供足够的电流给开发板使用。

29. 电源开关

这是开发板板载的电源开关（K1）。该开关用于控制整个开发板的供电。这是一个两段式拨动开关，拨到右边关闭开发板电源，整个开发板都将断电，电源指示灯（PWR）会随之熄灭。拨到右边打开开发板电源，整个板子开始供电，电源指示灯(PWR)点亮。

30. 5V 电源输入/输出

这是开发板板载的一组 5V 电源输入输出排针（2*3）（VOUT2），该排针用于给外部提供 5V 的电源，也可以用于从外部接 5V 的电源给板子供电。

同样大家在实验的时候可能经常会为没有 5V 电源而苦恼不已，正点原子充分考虑到了大家需求，有了这组 5V 排针，你就可以很方便的拥有一个简单的 5V 电源（USB 供电的时候，最大电流不能超过 500mA，外部供电的时候，最大可达 3000mA）。

31. 3.3V 电源输入/输出

这是开发板板载的一组 3.3V 电源输入输出排针（2*3）（VOUT1），用于给外部提供 3.3V 的电源，也可以用于从外部接 3.3V 的电源给板子供电。

大家在实验的时候可能经常会为没有 3.3V 电源而苦恼不已，有了 I.MX6U-ALPHA 开发板，你就可以很方便的拥有一个简单的 3.3V 电源（最大电流不能超过 3000mA）。

32. 3 路 USB HOST 接口

这是开发板板载的 3 路 USB HOST 接口，I.MX6U 有两个 USB 接口，正点原子的 I.MX6U-ALPHA 开发板通过 GL850 芯片将 I.MX6U 的 USB2 扩展成了 4 路 USB HOST，其中一路用于连接 4G 模块，另外 3 路作为 USB HSOT，用户可以通过这三路 USB HOST 接口连接 USB 鼠标、USB 键盘、U 盘等设备。

33. 引出的 IO 口

这是开发板 IO 引出端口 JP6，采用 2*16 排针，总共引出 31 个 IO 口。

34. 以太网接口 1(RJ45)

I.MX6U 有两个网络接口：ENET1 和 ENET2，这是 ENET1 网络接口，可以用来连接网线，实现网络通信功能。该接口使用 I.MX6U 内部的 MAC 控制器外加 PHY 芯片，实现 10/100M 网络的支持。

35. 以太网接口 2(RJ45)

这是开发板板载的以太网接口 2，也就是 I.MX6U 的 ENET2 网络接口。

36. RS232 接口（母）

这是开发板板载的另外一个 RS232 接口（COM3），通过一个标准的 DB9 母头和外部的串口连接。通过这个接口，我们可以连接带有串口的电脑或者其他设备，实现串口通信

37. RS485 接口

这是开发板板载的 RS485 总线接口（RS485），通过 2 个端口和外部 485 设备连接。这里提醒大家，RS485 通信的时候，必须 A 接 A，B 接 B。否则可能通信不正常

接下来，我们来看 I.MX6U 核心板的资源说明：

1. 核心板电源指示灯

这是核心板板载的一个蓝色 LED 灯，用于指示核心板供电是否正常，如果核心板供电正常

的话此灯就会点亮。

2. NAND/EMMC 存储芯片

这是核心板上板载的存储芯片，分为 NAND 和 EMMC 两种。对于 NAND 版本的核心板共有 256MB 和 512MB 两种容量的 NAND，型号分别为 MT29F2G08ABAEAWP-IT 或 MT29F4G08ABADAWP-IT，这两种型号的 NAND FLASH 工作温度范围都为工业级。EMMC 版本的核心板使用 8GB 的 EMMC，型号为 KLM8G1GET。

3. DDR3L 芯片

这是核心板板载的 DDR3L 芯片，NAND 版本核心板的 DDR3L 容量为 256MB，EMMC 版本的核心板的 DDR3L 容量为 512MB。型号分别为 NT5CC128M16JR-EK 和 NT5CC256M16EP-EK。如果要用于 UI 开发，那么最好选择 512MB 的 DDR3L，当然了，正点原子的 I.MX6U 核心板支持定制，具体定制方法请联系销售。

4. CPU

这是核心板的 CPU，型号为：MCIMX6Y2CVM05AB 或 MCIMX6Y2CVM08AB。这两款型号的外设功能都一模一样，只是 CPU 主频不同，MCIMX6Y2CVM05AB 主频为 528MHz，MCIMX6Y2CVM08AB 主频为 800MHz(实际 792MHz)。该芯片采用 Coretx-A7 内核，自带 32KB 的 L1 指令 Cache、32KB 的 L1 数据 Cache、128KB 的 L2Cache、集成 NEON 和 SIMDv2、支持硬件浮点(FPU)计算单元，浮点计算架构为 VFPv4-D32、1 个 RGB LCD 接口、2 个 CAN 接口、2 个 10M/100M 网络接口、2 个 USB OTG 接口(USB2.0)、2 路 ADC、8 个串口、3 个 SAI、4 个定时器、8 路 PWM、4 路 I2C 接口、4 路 SPI 接口、一路 CSI 摄像头接口、2 个 USDHC 接口，支持 4 位 SD 卡，最高可以支持 UHS-I SDR 104 模式，支持 1/4/8 位的 EMMC，最高可达 HS200 模式、一个外部存储接口、支持 16 位的 LPDDR2-800、DDR3-800 和 DDR3L-800、支持 8 位的 MLC/SLC NAND Flash，支持 2KB、4KB 和 8KB 页大小，以及 124 个通用 IO 口等。

5. 32.768KHz 晶振

这是一个无源的 32.768KHz 晶振，供 I.MX6U 内部 RTC 使用。

6. 24MHz 晶振

这是一个无源的 24MHz 晶振，供 I.MX6U 使用。

另外，I.MX6U 核心板的接口在底部，通过两个 2*30 的板对板端子（3710M 母座）组成，总共引出了 104 个 IO，通过这个接口，可以实现与 I.MX6U-ALPHA 底板对接。

5.2.2 软件资源说明

上面我们详细介绍了正点原子 I.MX6U-ALPHA 开发板的硬件资源。接下来，我们将向大家简要介绍一下 I.MX6U-ALPHA 开发板的软件资源。软件资源分为 3 部分：Linux 系统驱动软件资源、裸机例程、Linux 驱动例程，我们依次来看一下这三类软件资源的情况。关于 Linux 系统软件资源如表 5.2.2.1 所示：

类型	描述	备注
uboot	uboot 版本为 2016.03	提供源码。 支持 LCD 显示、支持 SD 卡和 EMMC、支持网络、支持 NAND Flash、支持环境变量修改等。
Linux 内核	内核版本为 4.1.15	提供源码
根文件系统 rootfs	提供 busybox、buildroot、yocto、ubuntu 这四种根文件系统及其制作方法	提供详细的制作教程

QT5 根文件系统	QT 版本为 5.6.1	提供详细的教程
交叉编译器	arm-linux-gnueaihf，版本 4.9.4	提供软件
系统烧写方法	MFGTOLL 和 SD 卡两种	提供详细的使用教程
LCD 驱	RGB LCD 驱动	提供源码
触摸	FT5xx6、GT9147 等电容触摸屏(仅限正点原子在售)	提供源码
485	RS485 驱动	提供源码
232	RS232 驱动	提供源码
CAN	CAN 驱动	提供源码
网络	PHY 为 LAN8720	提供源码
USB HOST	USB HUB 为 GL850	提供源码
USB OTG	USB 从机和主机	提供源码
4G 无线	ME3630 4G 模块	提供源码
按键 KEY	GPIO	提供源码
LED	GPIO	提供源码
音频	音频 DAC 为 WM8960	提供源码
SDIO WIFI	正点原子 RTL8189 模块	提供源码
GPS	正点原子 GPS 模块	提供源码
环境光传感器(IIC)	AP3216C，IIC 接口	提供源码
六轴传感器(SPI)	ICM20608，SPI 接口	提供源码
TF 卡/EMMC	USDHC 驱动	提供源码
摄像头	OV5640 驱动	提供源码
串口	UART1 驱动	提供源码
PWM 背光	LCD PWM 背光	提供源码
RTC	I.MX6U 内部 RTC	提供源码
USB WIFI	RTL8188	提供源码

表 5.2.2.1 开发板 Linux 系统软件资源

接下来看一下 I.MX6U-ALPHA 开发板的裸机例程，如表 5.2.2.2 所示：

编号	实验名字	编号	实验名字
1	leds	13	uart
2	ledc	14	printf
3	ledc_stm32	15	ddr3
4	ledc_sdk	16	lcd
5	ledc_bsp	17	rtc
6	beep	18	i2c
7	key	19	spi
8	clk	20	touchscreen
9	int	21	pwm_lcdbacklight
10	epit_timer		
11	key_filter		
12	highpreci_delay		

表 5.2.2.2 正点原子 I.MX6U-ALPHA 开发板裸机例程

从上表可以看出，正点原子的I.MX6U-ALPHA开发板裸机例程似乎不是很多，不像STM32、RT1052 那样六七十个裸机例程，这是因为嵌入式 Linux 和单片机的开发方式以及应用场合不同。单片机学名叫做 Microcontroller，也就是微控制器，主要用于控制相关的应用，因此单片机的外设都比较多，比如很多路的 IIC、SPI、UART、定时器等等。嵌入式 Linux 开发主要注重于高端应用场合，比如音视频处理、网络处理等等。比如一个机器人，高性能处理器加 Linux 系统(或者其他系统)作为机器人的大脑，主要负责接收各个传感器采集的数据然后对原始数据进行处理，得到下一步执行指令，这个往往需要很高的性能。当处理完成得到下一步要做的动作之后大脑就会将数据发给控制机器人各个关节电机的驱动控制器，这些驱动控制器一般都是单片机做的。所以大家在学习嵌入式 Linux 开发的时候一定不要深陷裸机，我们之所以讲解裸机是为了给嵌入式 Linux 打基础，让大家了解所使用的 SOC、了解 GCC 那一套工作流程，最终的目的都是为了嵌入式 Linux 做准备的。

看完裸机例程以后我们最后再来看一下正点原子为 I.MX6U-ALPHA 开发板准备的嵌入式Linux 驱动例程，如表 5.2.2.3 所示：

编号	实验名字	编号	实验名字
1	chrdevbase	13	irq
2	led	14	blockio
3	newchrled	15	noblockio
4	dtsled	16	asyncnoti
5	gpioled	17	platform
6	beep	18	dtsplatform
7	atomic	19	miscbeep
8	spinlock	20	input
9	semaphore	21	iic
10	mutex		spi
11	key		
12	timer		

表 5.2.2.3 正点原子 I.MX6U-ALPHA 开发板 Linux 驱动例程

因为有些外设驱动在 Linux 内核里面已经集成了，因此并没有编写独立的驱动，我们会在相应的章节里面对这些驱动进行讲解。关于正点原子 I.MX6U-ALPHA 开发板的软件资源就讲解到这里，软件资源我们也会持续更新的。

5.3 开发板底板原理图详解

5.3.1 核心板接口

I.MX6U-ALPHA 开发板采用底板+核心板的形式，I.MX6U-ALPHA 开发板底板采用 2 个2*30 的 3710F（公座）板对板连接器来同核心板连接，接插非常方便，底板上面的核心板接口原理图如图 5.3.1.1 所示：

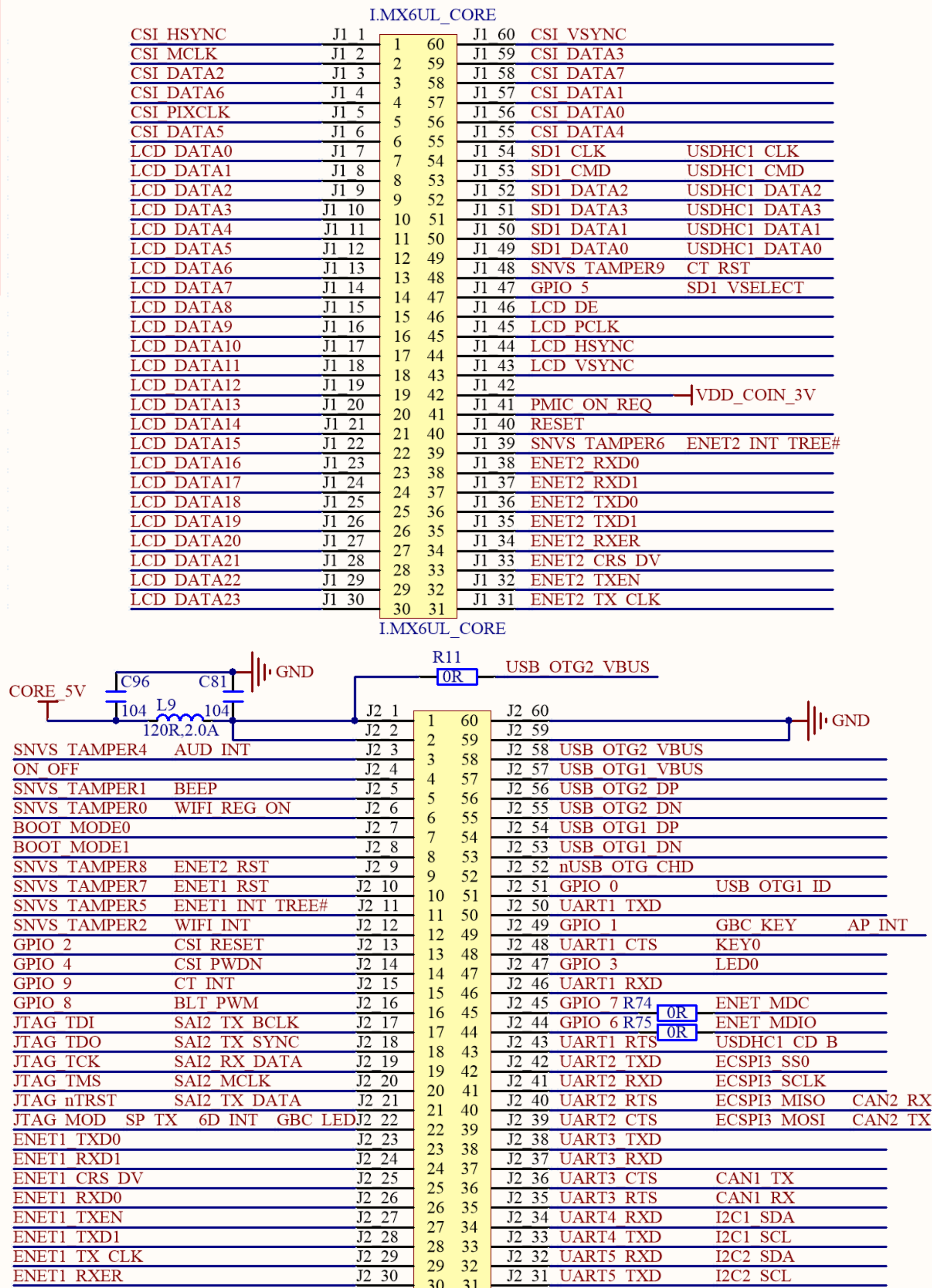


图 5.3.1.1 底板转接板接口部分原理图

图中的 J1 和 J2 就是底板上的转接板接口，由 2 个 2*30PIN 的 3710F 板对板公座组成，总共引出了核心板上面 105 个 IO 口，另外，还有：电源、PMIC_ON_REQ、ONOFF、USB、

VBAT、RESET 等信号。

5.3.2 引出 IO 口

I.MX6U-ALPHA 开发板底板上面，总共引出了 31 个 IO 口，如图 5.3.2.1 所示：

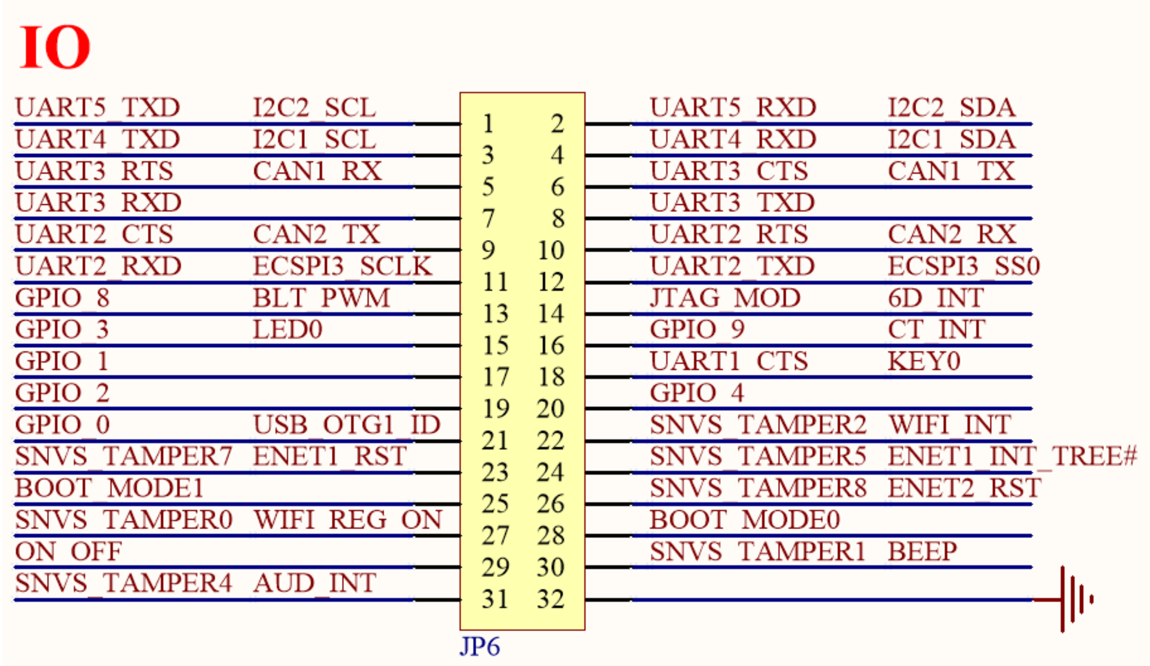


图 5.3.2.1 引出 IO 口

图中 JP6 就是引出的 IO，一共 31 个 IO，外加一个 GND。Cortex-A 系列的板子首要任务就是保证板子的稳定性，然后再考虑 IO 的引出。所以相比于 STM32 这种单片机，I.MX6U-ALPHA 开发板引出的 IO 就要少很多了。

5.3.3 USB 串口/串口 1 选择接口

I.MX6U-ALPHA 开发板板载的 USB 串口和 I.MX6U 的串口是通过 JP5 连接起来的，如图 5.3.3.1 所示：

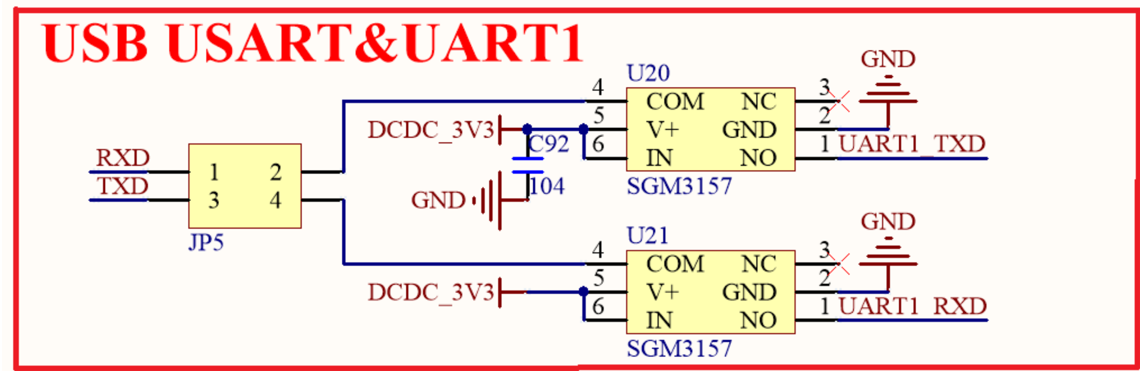


图 5.3.3.1 USB 串口/串口 1 选择接口

图中 TXD/RXD 是相对 CH340C 来说的，也就是 USB 串口的发送和接受脚。而 UART1_RXD 和 UART1_TXD 则是相对于 I.MX6U 来说的。这样，通过对接，就可以实现 USB 串口和 I.MX6U

的串口通信了。图 5.3.3.1 中的 U20 和 U21 是 SGM3157 是模拟开关，在底板掉电以后将 I.MX6U 的 UART1_TXD 和 UART1_RXD 这两个 IO 与 CH340C 的 TXD 和 RXD 断开，因为 CH340C 的 TXD 和 RXD 这两个 IO 带有微弱的 3.3V 电压，如果不断开的话会将这微弱的 3.3V 电压引入到核心板上，可能会影响到启动。

5.3.4 RGB LCD 模块接口

ALPHA 底板载了 RGB LCD 接口，此部分电路如图 5.3.4.1 所示：

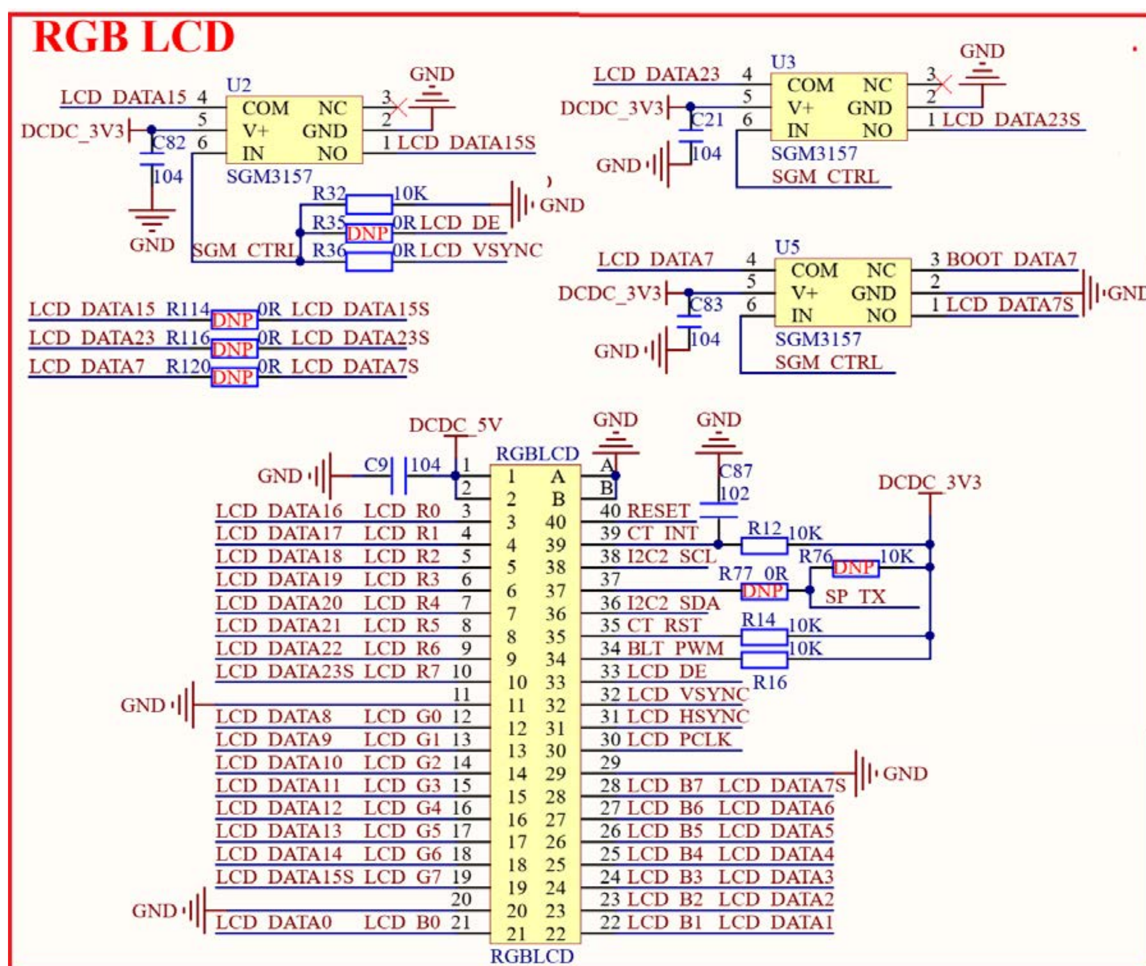


图 5.3.4.1 RGB LCD 接口

图中，RGBLCD 就是 RGB LCD 接口，采用 RGB888 数据格式，并支持触摸屏（支持电阻屏和电容屏）。该接口仅支持 RGB 接口的液晶（不支持 MCU 接口的液晶），目前正点原子的 RGB 接口 LCD 模块有：4.3 寸（ID:4342，480*272 和 ID:4384，800*480）、7 寸(ID:7084，800*480 和 ID:7016，1024*600)和 10 寸(ID:1018，1280*800)等尺寸可选。

图中 3 个 SGM3157 模拟开关，用于控制来自 I.MX6U 的 LCD_DATA23(LCD_R7)、LCD_DATA15(LCD_G7)和 LCD_DATA7(LCD_B7)和来自 RGBLCD 屏的 LCD_DATA23S、LCD_DATA15S 和 LCD_DATA7S 的通断。这是因为这几个信号有用来设置 I.MX6U 的 BOOT_CFG4[7]/BOOT_CFG2[7]/BOOT_CFG1[7]，同时又是 RGBLCD 屏的 ID 信号，因此他们存在冲突。如果不加切换，在启动的时候，I.MX6U 就可能读到错误的启动配置信息，从而导致启动失败（不运行代码）。加这三个模拟开关，就是为了让 I.MX6U 在启动的时候可以正常读取 BOOT_CFG4[7]/BOOT_CFG2[7]/BOOT_CFG1[7]的值，同时在启动后，用户代码又可以读取正

确的 RGBLCD ID 值。互不影响。三个 SGM3157 的使能信号默认都是由 LCD_VSYNC 控制（刚好满足 LCD 时序）。

图中的 I2C2_SCL 和 I2C2_SDA 为 I2C2 的两根数据线，分别连接到 UART5_TXD 和 UART5_RXD 这两个 IO 上。BLT_PWM 是 LCD 的背光控制 IO，连接在 I.MX6U 的 GPIO1_IO8 上，用于控制 LCD 的背光。液晶复位信号 RESET 则是直接连接在开发板的复位按钮上，和 MCU 共用一个复位电路。

5.3.5 复位电路

I.MX6U 开发板的复位电路如图 5.3.5.1 所示：

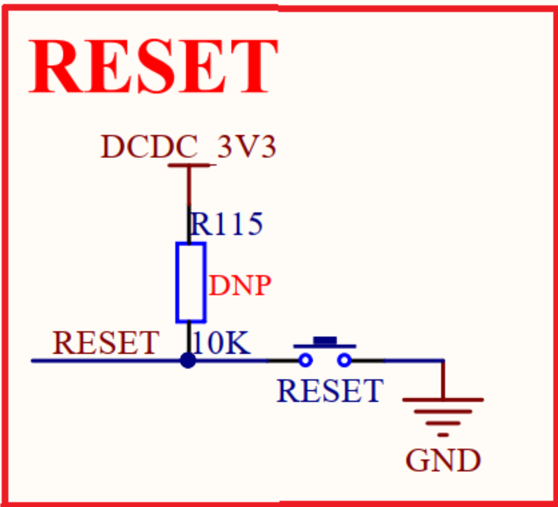


图 5.3.5.1 复位电路

因为 I.MX6U 是低电平复位的，所以我们设计的电路也是低电平复位的

5.3.6 启动模式设置接口

I.MX6U 开发板的启动模式设置端口电路如图 5.3.6.1 所示：

BOOT

SW

D1	D2	D3	D4	D5	D6	D7	D8	BOOT DEVICE
OFF	ON	OFF	OFF	OFF	OFF	OFF	OFF	USB
ON	OFF	OFF	OFF	OFF	OFF	ON	OFF	MicroSD
ON	OFF	ON	OFF	OFF	ON	ON	OFF	EMMC
ON	OFF	OFF	OFF	ON	OFF	OFF	ON	NAND

DCDC_3V3
R88 10K 1 — 16 BOOT_MODE1
R89 10K 2 — 15 BOOT_MODE0
R90 4.7K 3 — 14 LCD_DATA11
R91 4.7K 4 — 13 LCD_DATA3
R93 4.7K 5 — 12 LCD_DATA4
R95 4.7K 6 — 11 LCD_DATA5
R97 4.7K 7 — 10 LCD_DATA6
R98 4.7K 8 — 9 BOOT_DATA7
OFF ON
BOOT_CFG

图 5.3.6.1 启动模式设置接口

I.MX6U 有多种启动方式，并且支持从多重不同的设备启动，关于 I.MX6U 的详细启动方式请参考《第九章 I.MX6U 启动方式详解》。

5.3.7 VBAT 供电接口

I.MX6U-ALPHA 开发板的 VBAT 供电电路如图 5.3.7.1 所示：

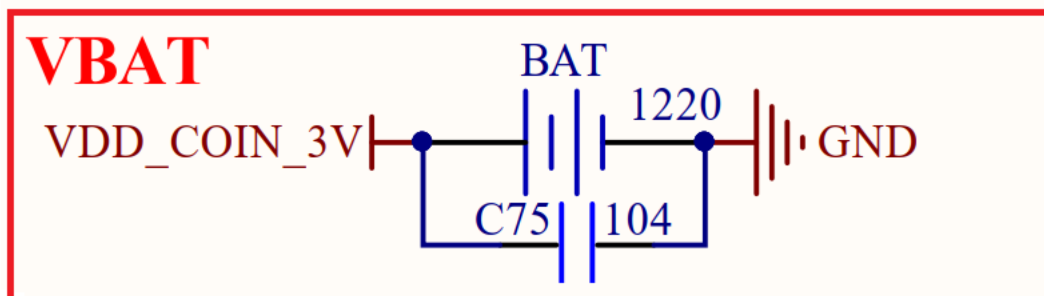


图 5.3.7.1 启动模式设置接口

上图的 VDD_COIN_3V 通过核心板上的 BAT54C，接 VDD_SNVS_IN 脚，从而给核心板的 SNVS 区域供电。这部分原理图在核心板上，如图 5.3.7.2 所示：

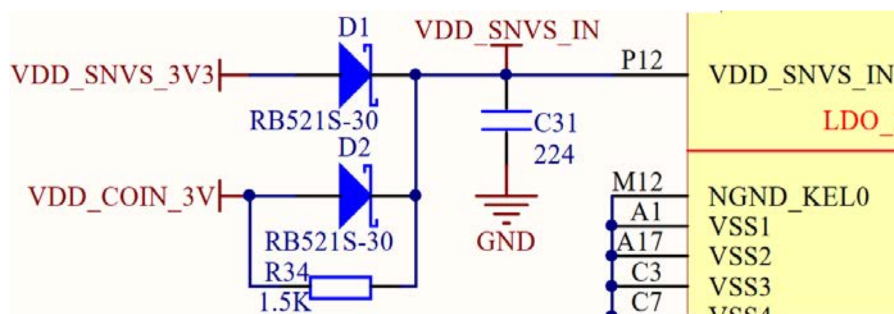


图 5.3.7.2 核心板 VBAT 供电原理图

如图 5.3.7.2 所示，VDD_SNVS_IN 使用 VDD_COIN_3V（接 CR1220 电池）和 VDD_SNVS_3V3 混合供电的方式，在有外部电源（VDD_SNVS_3V3）的时候，CR1220 不给 VDD_SNVS_IN 供电，而在外部电源断开的时候，则由 CR1220 给其供电。这样，VDD_SNVS_IN 总是有电的，以保证 RTC 的走时。

5.3.8 RS232 串口

I.MX6U-ALPAH 开发板板载了一个母头的 RS232 接口，电路原理图如图 5.3.8.1 所示：

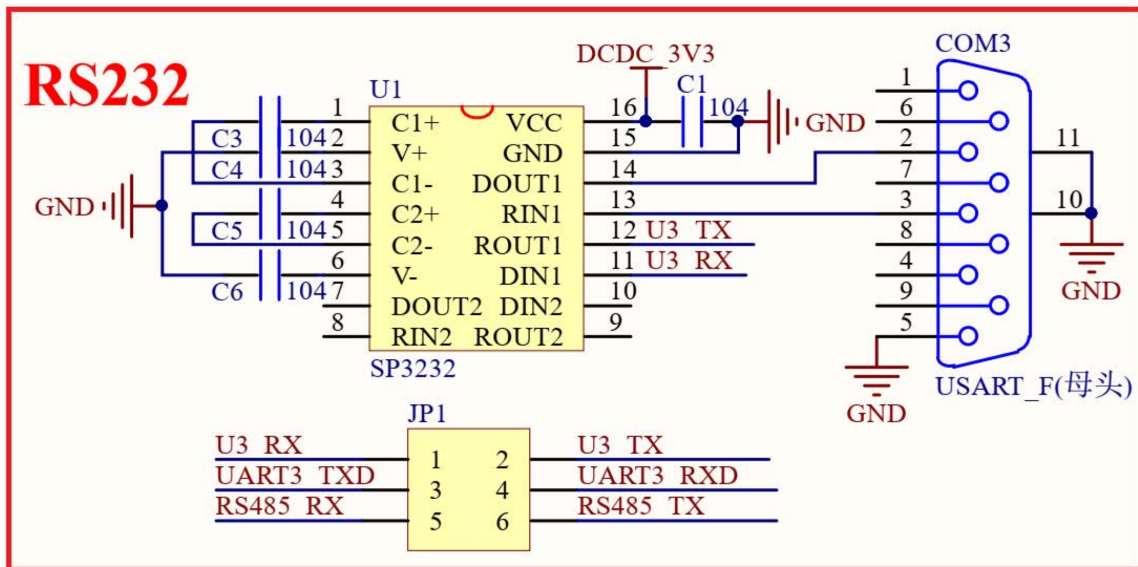


图 5.3.8.1 RS232 串口

因为 RS232 电平不能直接连接到 I.MX6U，所以需要一个电平转换芯片。这里我们选择的是 SP3232(也可以用 MAX3232)来做电平转接，同时图中的 JP1 用来实现 RS232(UART3)/RS485 的选择。所以这里的 RS232/RS485 都是通过串口 3 来实现的。图中 RS485_TX 和 RS485_RX 信号接在 SP3485 的 DI 和 RO 信号上。

5.3.9 RS485 接口

I.MX6U-ALPAH 开发板板载的 RS485 接口电路如图 5.3.9.1 所示：

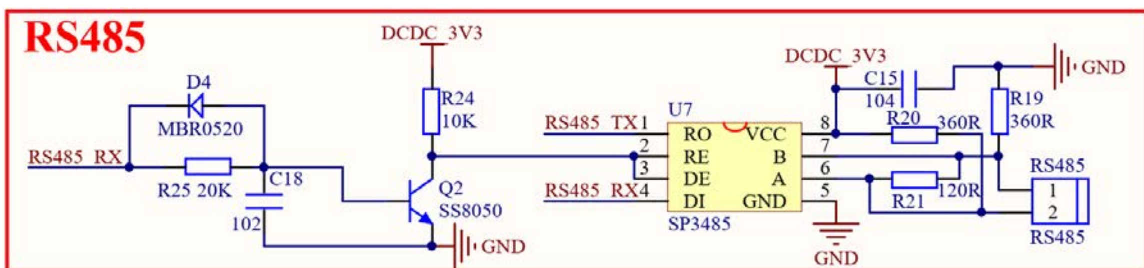


图 5.3.9.1 RS485 接口

RS485 电平也不能直接连接到 I.MX6U，同样需要电平转换芯片。这里我们使用 SP3485 来做 485 电平转换，其中 R21 为终端匹配电阻，而 R19 和 R20，则是两个偏置电阻，以保证静默状态时 485 总线维持逻辑 1。

RS485_RX/RS485_TX 连接在 JP1 上面，通过 JP1 跳线来选择是否连接在 I.MX6U 上面，SP3485 的 RE 引脚连接通过一系列的电路连接到了 RS485_RX 引脚上，这样就可以通过 RS485_RX 引脚来控制 RS485 的接收和发送状态，完全将 RS485 当做一个串口来使用。

5.3.10 CAN 接口

正点原子 I.MX6U-ALPHA 开发板板载的 CAN 接口电路如图 5.3.10.1 所示：

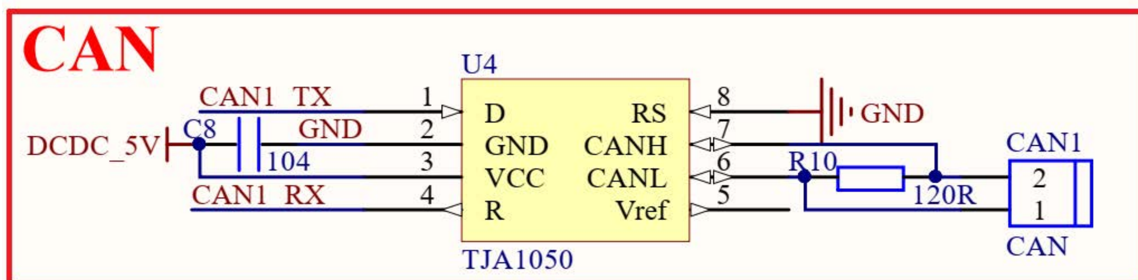


图 5.3.10.1 CAN 接口电路

CAN 总线电平也不能直接连接到 I.MX6U，同样需要电平转换芯片。这里我们使用 TJA1050 来做 CAN 电平转换，其中 R10 为终端匹配电阻。

CAN1_TX/CAN1_RX 直接连接在 I.MX6U 的 UART1_CTS/UART1_RTS 上面。

5.3.11 USB HUB 接口

正点原子 I.MX6U-ALPHA 开发板板载了一颗一扩四的 USB HUB 芯片，用于将 I.MX6U 的 USB2 扩展为 4 个 USB HOST 接口，如图 5.3.15.3 所示：

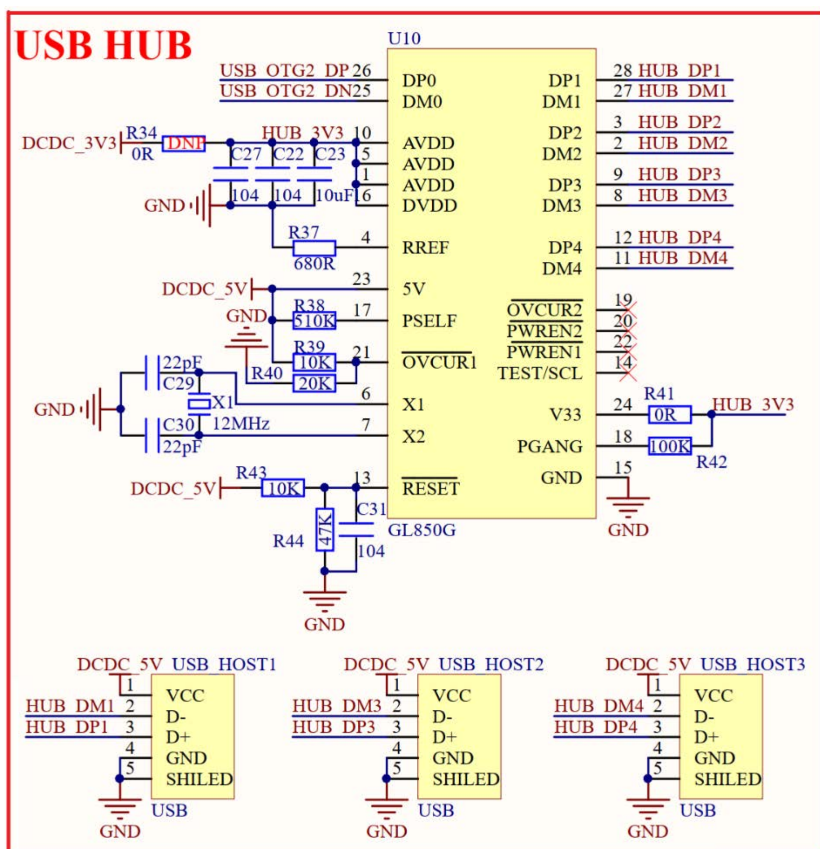


图 5.3.11.3 USB HUB 接口电路

I.MX6U 带有两个 USB 接口，但是对于 Linux 应用来说两个 USB 太少了，如果我们要连接鼠标、键盘、U 盘等设备的时候两个 USB 口完全不够用。因此 I.MX6U-ALPHA 开发板通过 GL850G 芯片将 I.MX6U 的 USB2 外扩出了 4 个 USB HOST 接口，其中有一路(UHB_USB2)外接了 4G 模块，因此提供给用户的用户的就只剩下了 3 个 USB HOST 接口，如果 3 个 USB HOST 还不够用的话就可以外接一个 USB HUB。

5.3.12 USB OTG 接口

I.MX6U-ALPHA 也有一路 USB OTG 接口，USB OTG 接口使用了 I.MX6U 的 USB1，USB OTG 接口如图 5.3.12.1 所示：

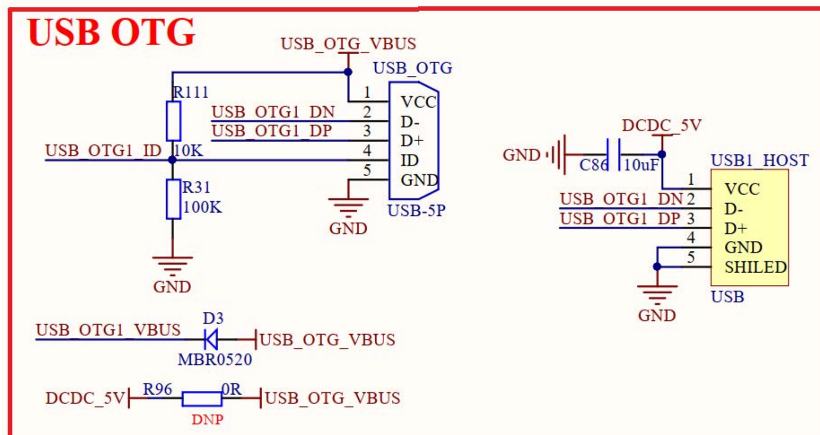


图 5.3.12.1 USB OTG 接口

图中的 USB_OTG 就是 USB SLAVE 接口，可以将开发板作为 USB 从机，比如通过 USB 进行系统烧写等。右侧的 USB1_HOST 是 USB HOST 接口，可以将开发板作为 USB 主机，这样就可以外接 USB 键盘/鼠标、U 盘等设备。这里正点原子将 USB OTG 的 SLAVE 和 HOST 接口都做到了开发板上，这样大家就不需要再额外去购买一个 USB OTG 线了，方便大家开发。

5.3.13 光环境传感器

I.MX6U-ALPHA 开发板板载了一个光环境传感器，可以用来感应周围光线强度、接近距离和红外线强度等，该部分电路如图 5.3.13.1 所示：

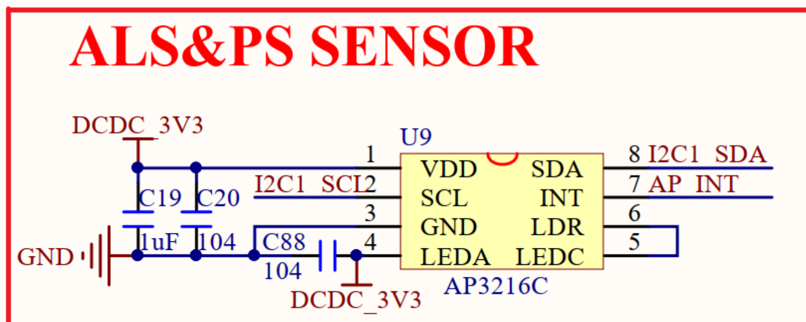


图 5.3.13.1 光环境传感器电路

图中的 U9 就是光环境传感器：AP3216C，它集成了光照强度、近距离、红外三个传感器功能于一身，被广泛应用于各种智能手机。该芯片采用 IIC 接口，连接在 I.MX6U 的 I2C1 接口上，IIC_SCL 和 IIC_SDA 分别连接在 UART4_TXD 和 UART4_RXD 上，AP_INT 是其中断输出脚，连接在 I.MX6U 的 GOIO1_IO01 上。

5.3.14 六轴传感器

I.MX6U-ALIPHA 开发板板载了一个六轴传感器，电路如图 5.3.14.1 所示：

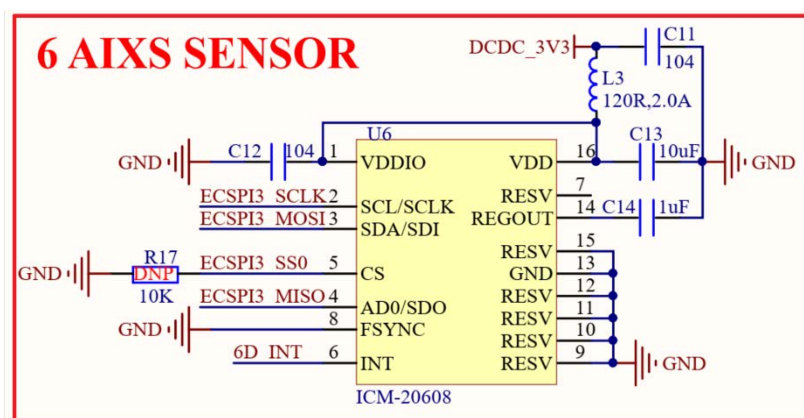


图 5.3.14.1 六轴传感器

六轴传感器芯片型号为：ICM20608，该芯片内部集成了：三轴加速度传感器和三轴陀螺仪，这里我们使用 SPI 接口来访问。

ICM20608 支持 IIC 和 SPI 两种接口，I.MX6U-ALPHA 开发板使用 SPI 接口，目的是为了在开发板上防一个 SPI 外设，学习 Linux 下的 SPI 驱动开发(因为笔者购买过很多 Linux 开发板，发现基本都没有 SPI 外设，不方便学习 SPI 驱动)。ICM20608 通过 SPI 接口连接到 I.MX6U 的 ECSPI3 接口上，SCLK、SDI、CS 和 SDO 分别连接到 I.MX6U 的 UART2_RXD(ECSPI3_SCLK)、UART2_CTS(ECSPI3_MOSI)、UART2_TXD(ECSPI3_SS0)和 UART2_RTS(ECSPI3_MISO)。

5.3.15 LED

I.MX6U-ALPHA 开发板板载总共有 2 个 LED，其原理图如图 5.3.15.1 所示：

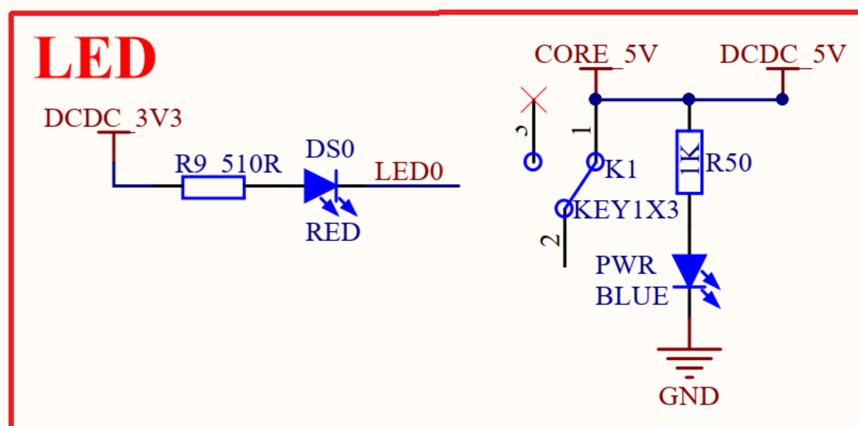


图 5.3.15.1 LED

其中右侧的 PWR BLUE 是系统电源指示灯，为蓝色。DS0 为用户 LED 灯，连接在 I.MX6U 的 GPIO1_IO03 上，此灯为红色。

5.3.16 按键

I.MX6U-ALPHA 开发板板载 1 个输入按键，其原理图如图 5.3.16.1 所示：

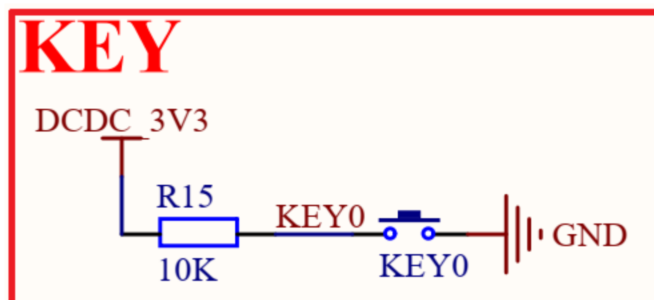


图 5.3.16.1 输入按键

KEY0 用作普通按键输入，分别连接 I.MX6U 的 UART1_CTS 引脚上，这里使用外部 10K 上拉电阻。

5.3.17 摄像头模块接口

I.MX6U-ALPHA 开发板板载了一个摄像头模块接口，连接在 I.MX6U 的硬件摄像头接口（CSI）上面，其原理图如图 5.3.17.1 所示：

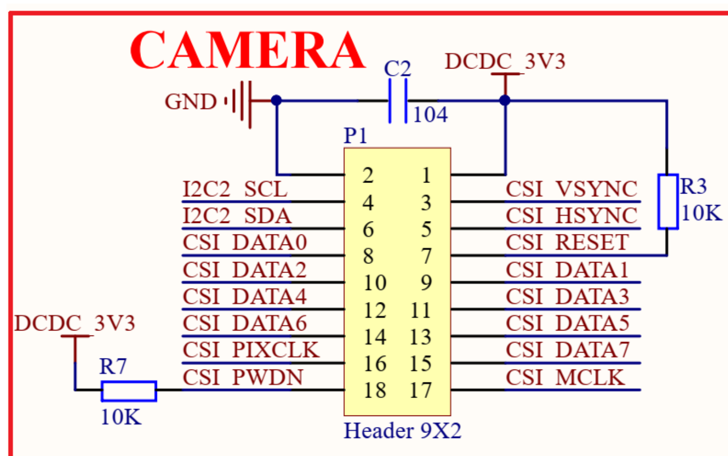


图 5.3.17.1 摄像头模块接口

图中 P1 接口可以用来连接正点原子摄像头模块。其中，I2C2_SCL 和 I2C2_SDA 是摄像头的 SCCB 接口，分辨连接在 I.MX6U 的 UART5_TXD 和 UART5_RXD 引脚上。CSI_RESET 和 CSI_PWDN 这 2 个信号是不属于 I.MX6U 硬件摄像头接口的信号，通过普通 IO 控制即可，这两个线分别接在 I.MX6U 的 GPIO1_IO02 和 GPIO1_IO04。

此外，CSI_VSYNC/CSI_HSYNC/CSI_D0/CSI_D1/CSI_D2/CSI_D3/ CSI_D4/CSI_D5/CSI_D6/CSI_D7/CSI_PCLK/CSI_MCLK 等信号，接 I.MX6U 的硬件摄像头接口。

5.3.18 有源蜂鸣器

I.MX6U-ALPHA 开发板板载了一个有源蜂鸣器，其原理图如图 5.3.18.1 所示：

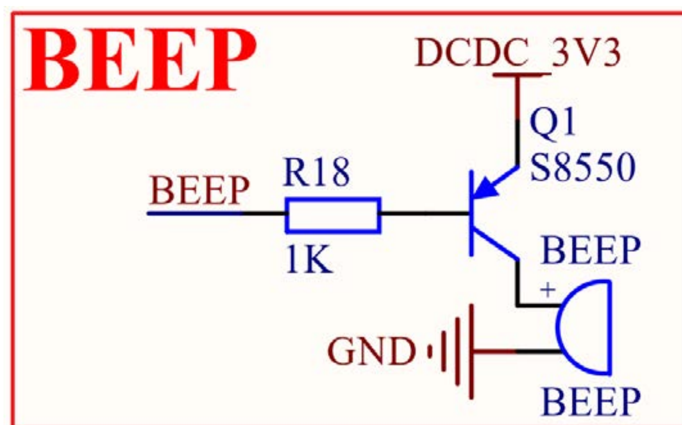


图 5.3.18.1 有源蜂鸣器

有源蜂鸣器是指自带了震荡电路的蜂鸣器，这种蜂鸣器一接上电就会自己震荡发声。而如果是无源蜂鸣器，则需要外加一定频率（2~5Khz）的驱动信号，才会发声。这里我们选择使用有源蜂鸣器，方便大家使用。

BEEP 信号直接连接在 I.MX6U 的 SNVS_TAMPER1 引脚上，可以通过控制此引脚来控制蜂鸣器开关。

5.3.19 TF 卡接口

正点原子 ALPHA 开发板板载了一个 SD 卡（大卡）接口，其原理图如图 5.3.24.1 所示：

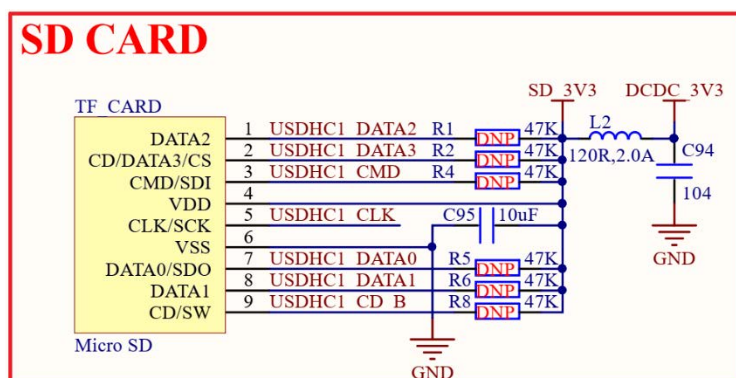


图 5.3.19.1 SD 卡接口

图中 SD_CARD 为 TF 卡接口，该接口在开发板的底面。

TF 卡采用 4 位 uSDHC 方式驱动，非常适合需要高速存储的情况。图中：USDHC1_DATA0~DATA3/USDHC1_CLK/USDHC1_CMD 分别连接在 I.MX6U 的 SD1_DATA0~DATA3/SD1_CLK/SD1_CMD 引脚上。 USDHC1_CD_B 是 TF 卡检测引脚，用于检测 TF 卡或 SDIO WIFI 插拔过程，连接到 I.MX6U 的 UART1_RTS 引脚上。注意：TF 卡接口和 SDIO WIFI 接口公用一个 SDIO，因此 TF 卡和 SDIO 不能同时使用！！

5.3.20 SDIO WIFI 接口

I.MX6U-ALPHA 开发板板子一个 SDIO WIFI 接口，如图 5.3.20.1 所示：

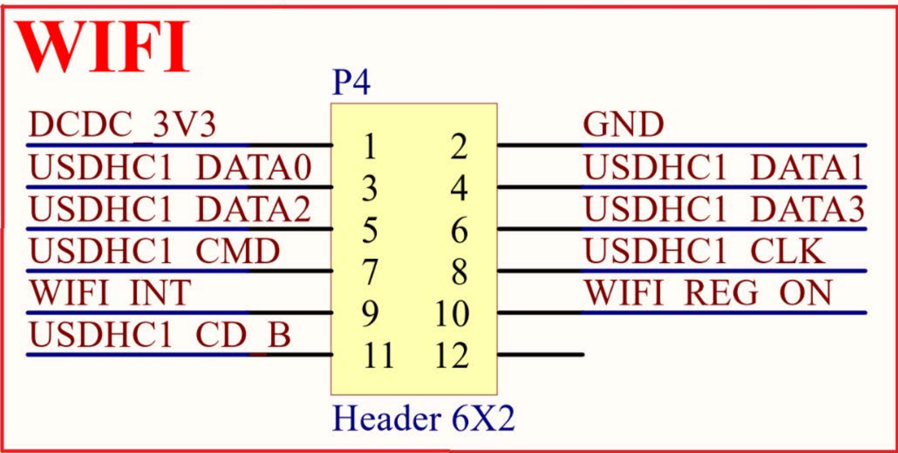


图 5.3.20.1 SDIO WIFI 接口

SDIO WIFI 接口用于连接正点原子出品的 RTL8189 SDIO WIFI 模块，USDHC1_DATA0~DATA3/USDHC1_CLK/USDHC1_CMD 分别连接到 I.MX6U 的 I.MX6U 的 SD1_DATA0~DATA3/SD1_CLK/SD1_CMD 引脚上。WIFI_INT 和 WIFI_REG_ON 连接到 I.MX6U 的 SNVS_TAMPER2 和 SNVS_TAMPER0 引脚上。

注意：TF 卡接口和 SDIO WIFI 接口公用一个 SDIO，因此 TF 卡和 SDIO 不能同时使用!!

5.3.21 4G 模块接口

I.MX6U-ALPHA 开发板板载 4G Mini PCIE 接口，如图 5.3.21.1 所示：

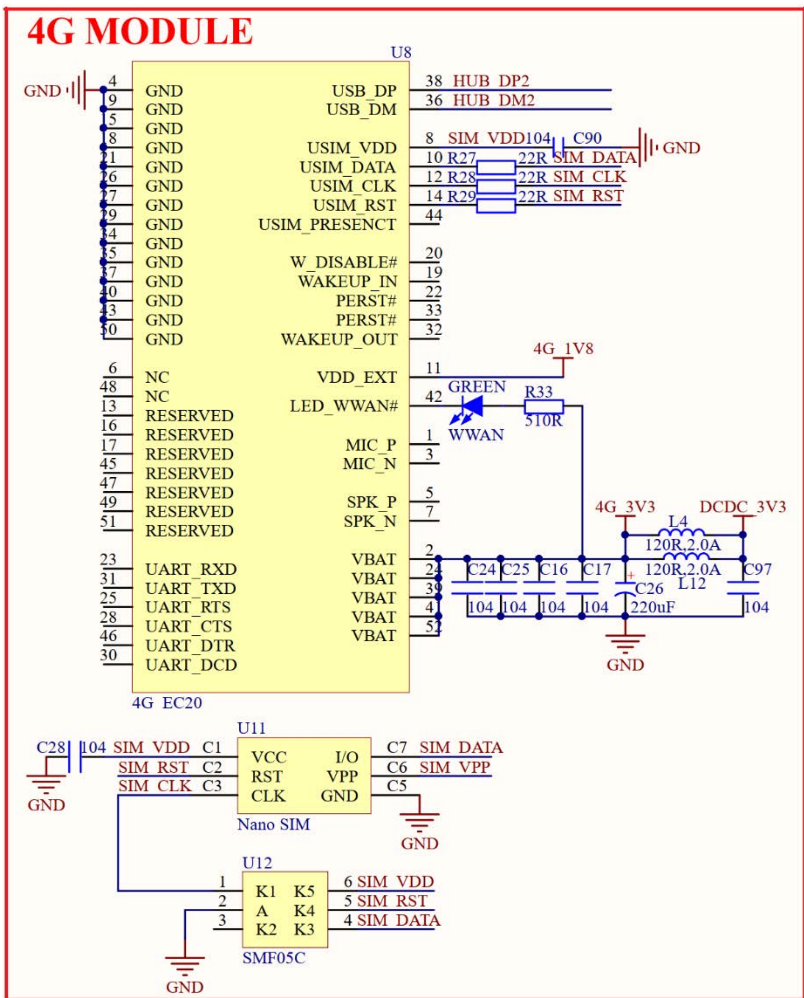


图 5.3.21.1 4G 模块

U8 就是 Mini PCIE 接口的 4G 模块座子，用于连接 Mini PCIE 接口的 4G 模块，比如高新兴的 ME3630 模块。U11 是 Nano SIM 卡座，用于插入 Nano SIM 卡。4G 模块虽然采用 Mini PCIE 接口，但是实际走的 USB 接口，这里连接到了 GL850G 扩展出来的一个 USB HOST 接口上(USB_HUB2)。

5.3.22 ATK 模块接口

I.MX6U-ALPHA 开发板板载了 ATK 模块接口，其原理图如图 5.3.22.1 所示：

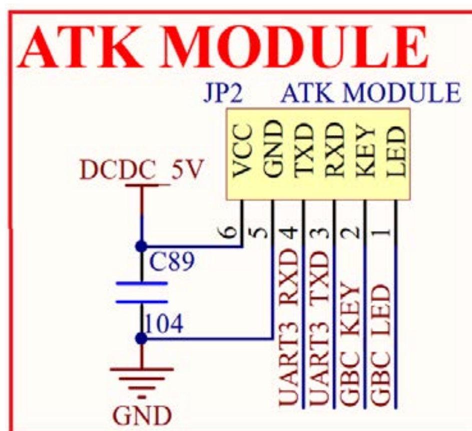


图 5.3.22.1 ATK 模块接口

如图所示，JP2 是一个 1*6 的排座，可以用来连接正点原子推出的一些模块，比如：蓝牙串口模块、GPS 模块、MPU6050 模块、激光测距模块、手势识别模块和 RGB 彩灯模块等。有了这个接口，我们连接模块就非常简单，插上即可工作。

图中：UART3_RXD/UART3_TXD 连接到了 I.MX6U 的 UART3 上，和 RS232、RS485 共用一个串口，在使用 ATK 接口的时候需要将 JP1 跳线帽全部拔掉，防止 RS232 和 RS485 干扰到模块。而 GBC_KEY 和 GBC_LED 则分别连接在 I.MX6U 的 GPIO1_IO01 和 JTAG_MOD 这两个引脚上。**特别注意：** GBC_LED/ GBC_KEY 和 AP_INT/6D_INT 共用 GPIO1_IO01 和 JTAG_MOD。

5.3.23 以太网接口（RJ45）

I.MX6U-ALPHA 开发板板载了两个以太网接口(RJ45)，分别为 ENET1 和 ENET2，其中 ENET1 网口的原理图如图 5.3.23.1 所示：

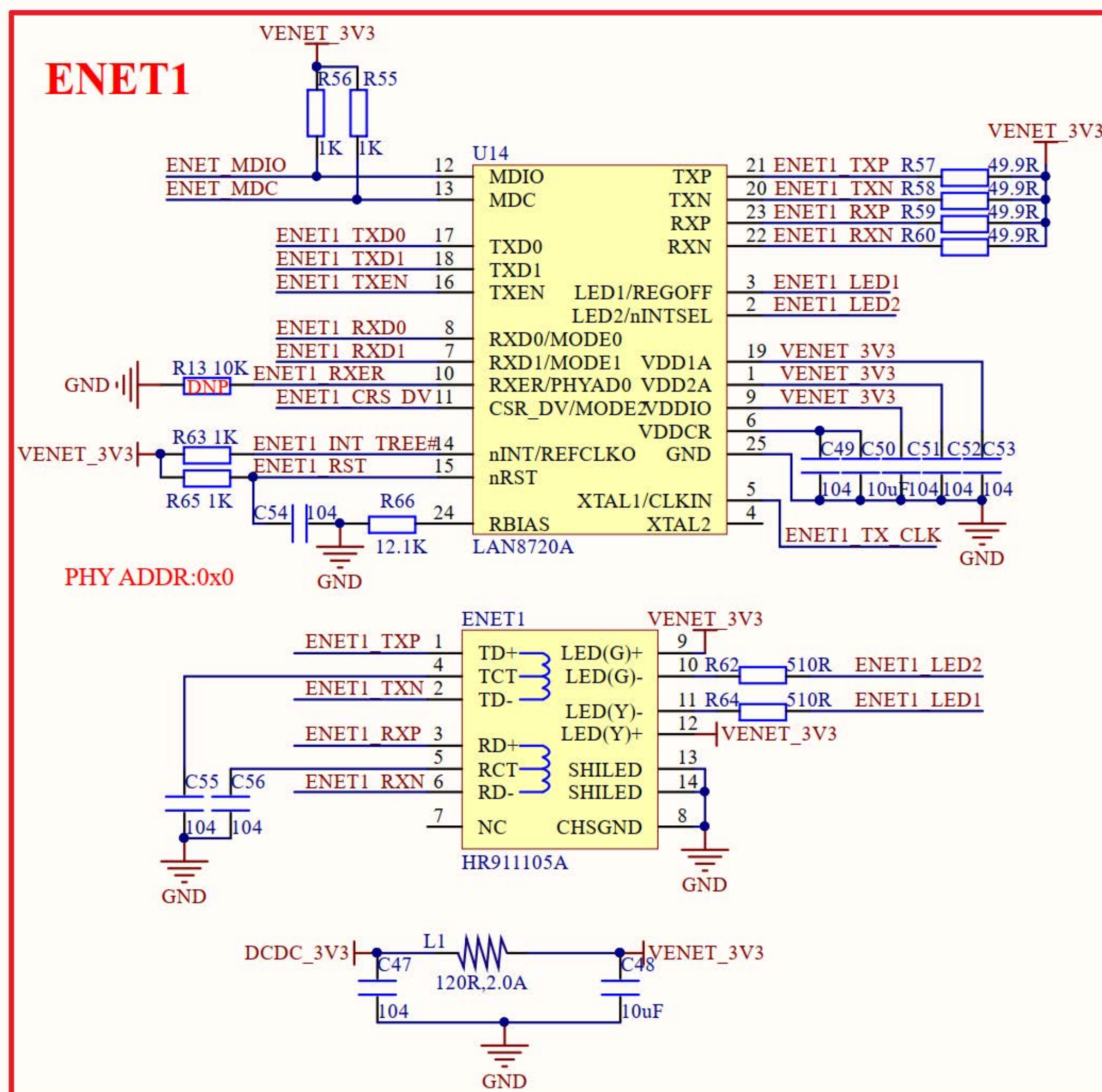


图 5.3.23.1 ENET1 接口电路

ENET2 网络接口原理图如图 5.3.23.2 所示：

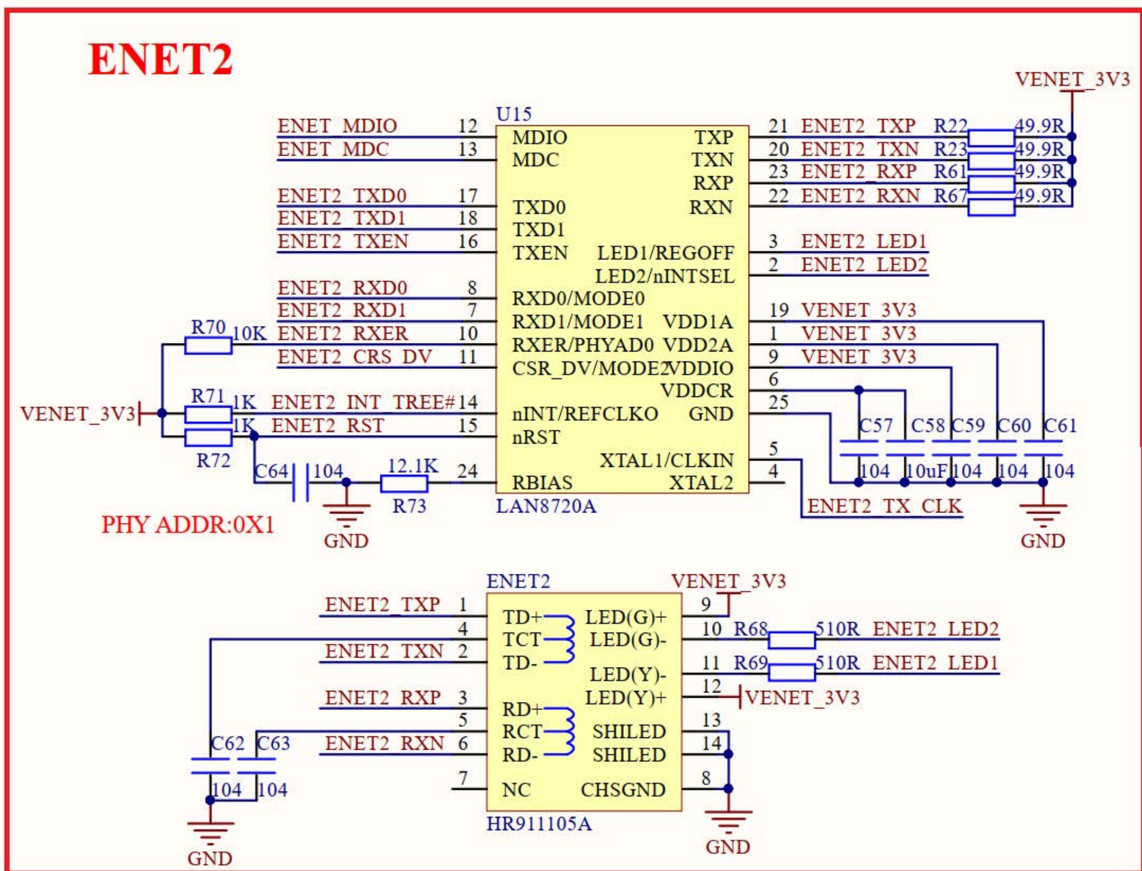


图 5.3.23.2 ENET2 网络接口

I.MX6U 内部自带两个网络 MAC 控制器，每个网络 MAC 需要外加一个 PHY 芯片，即可实现网络通信功能。这里我们选择的是 LAN8720A 这颗芯片作为 I.MX6U 的 PHY 芯片，该芯片采用 RMII 接口与 I.MX6U 通信，占用 IO 较少，且支持 auto mdix（即可自动识别交叉/直连网线）功能。板载两个自带网络变压器的 RJ45 头（HR91105A），一起组成两个 10M/100M 自适应网卡。

对于 ENET1 和 ENET2 共用 ENET_MDIO 和 ENET_MDC 这两根线，这两根线连接到了 I.MX6U 的 GPIO1_IO06 和 GPIO1_IO07 这两个 IO 上。ENET1 和 ENET2 都有一个复位引脚，分别为 ENET1_RSET 和 ENET2_RESET，这两个 IO 分别连接到了 I.MX6U 的 SNVS_TAMPER7/SNVS_TAMPER8 这两个引脚上。

5.3.24 SAI 音频编解码器

I.MX6U 开发板板载 WM8960 高性能音频编解码芯片，其原理图如图 5.3.24.1 所示：

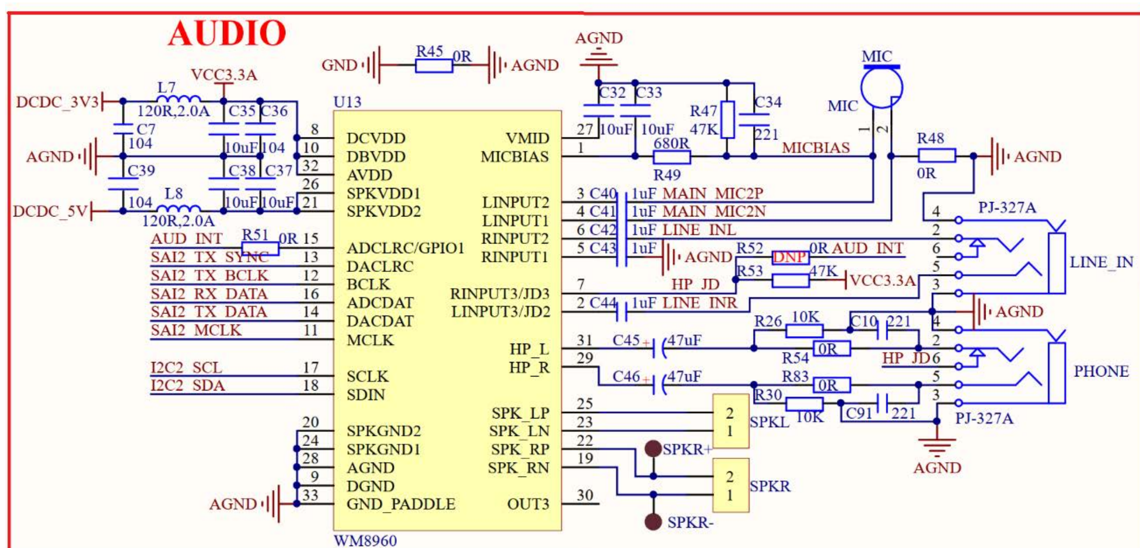


图 5.3.24.1 SAI 音频编解码芯片

WM8960 是一颗低功耗、高性能的立体声多媒体数字信号编解码器。该芯片内部集成了 24 位高性能 DAC&ADC，并且自带段 EQ 调节，支持 3D 音效等功能。不仅如此，该芯片还结合了立体声差分麦克风的前置放大与扬声器、耳机和差分、立体声线输出的驱动，减少了应用时必需的外部组件，直接可以驱动耳机（16Ω@40mW）和喇叭（8Ω/1W），无需外加功放电路。

图中，SPK-和 SPK+连接了一个板载的 8Ω 1W 小喇叭（在开发板背面）。MIC 是板载的咪头，可用于录音机实验，实现录音。PHONE 是 3.5mm 耳机输出接口，可以用来插耳机。LINE_IN 则是线路输入接口，可以用来外接线路输入，实现立体声录音。

该芯片采用 SAI 与 I.MX6U 的 SAI 接口连接，图中：SAI2_TX_SYNC/SAI2_TX_BCLK/SAI2_RX_DATA/SAI2_TX_DATA/SAI2_MCLK/ 分别接在 MCU 的：JTAG_TDO/JTAG_TDI/JTAG_TCK/JTAG_nTRST/JTAG_TMS 上。**特别注意：**WM8960 和 JTAG 共用了很多信号，所以 WM8960 和 JTAG 接口不能同时使用！并且，经过实测，WM8960 会干扰到 JTAG，如果要使用 JTAG 那么就必须将与 WM8960 复用的这几根线断开！这也是为什么正点原子 I.MX6U-ALPHA 开发板开发板上面没有留 JTAG 接口的原因。

WM8960 需要一个 I2C 接口去配置，这里使用 I.MX6U 的 I2C2，I2C2_SCL 和 I2C2_SDA 分别连接到了 I.MX6U 的 UART5_TXD 和 UART5_RXD 这两个引脚上。

5.3.25 电源

I.MX6U-ALPHA 开发板板载的电源供电部分，其原理图如图 5.3.25.1 所示：

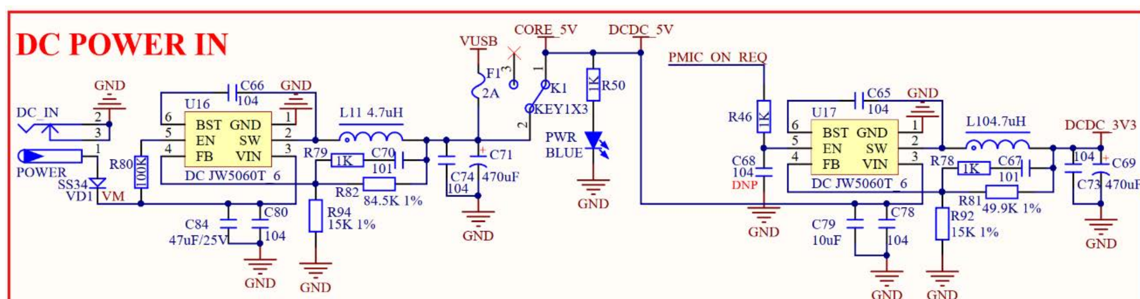


图 5.3.25.1 电源

图中，总共有 2 个稳压芯片：U16/U17，DC_IN 用于外部直流电源输入，经过 U16 DC-DC

芯片转换为 5V 电源输出，其中 VD1 是防反接二极管，避免外部直流电源极性搞错的时候，烧坏开发板。K1 为开发板的总电源开关，F1 为 2A 自恢复保险丝，用于保护 USB。U17 为 3.3V 稳压芯片，给开发板提供 3.3V 电源。

这里还有 USB 供电部分没有列出来，其中 VUSB 就是来自 USB 供电部分，详见 5.3.26 节。

5.3.26 电源输入输出接口

I.MX6U-ALPHA 开发板板载了两组简单电源输入输出接口，其原理图如图 5.3.27.1 所示：

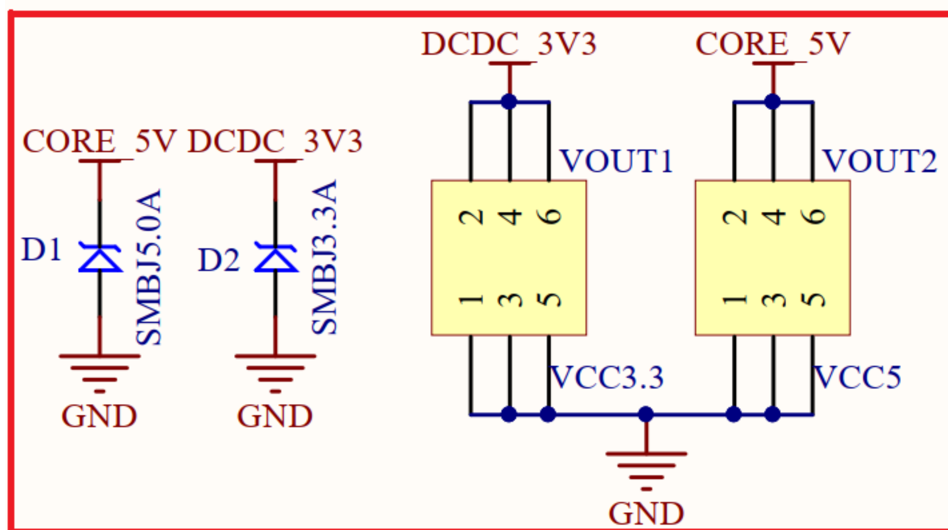


图 5.3.26.1 电源

图中，VOUT1 和 VOUT2 分别是 3.3V 和 5V 的电源输入输出接口，有了这 2 组接口，我们可以通过开发板给外部提供 3.3V 和 5V 电源了，虽然功率不大（最大 1000ma），但是一般情况都够用了，大家在调试自己的小电路板的时候，有这两组电源还是比较方便的。同时这两组端口，也可以用来由外部给开发板供电。

图中 D1 和 D2 为 TVS 管，可以有效避免 VOUT 外接电源/负载不稳的时候（尤其是开发板外接电机/继电器/电磁阀等感性负载的时候），对开发板造成的损坏。同时还能一定程度防止外接电源接反，对开发板造成的损坏。

5.3.27 USB 串口

I.MX6U-ALPHA 开发板板载了一个 USB 串口，其原理图如图 5.3.27.1 所示：

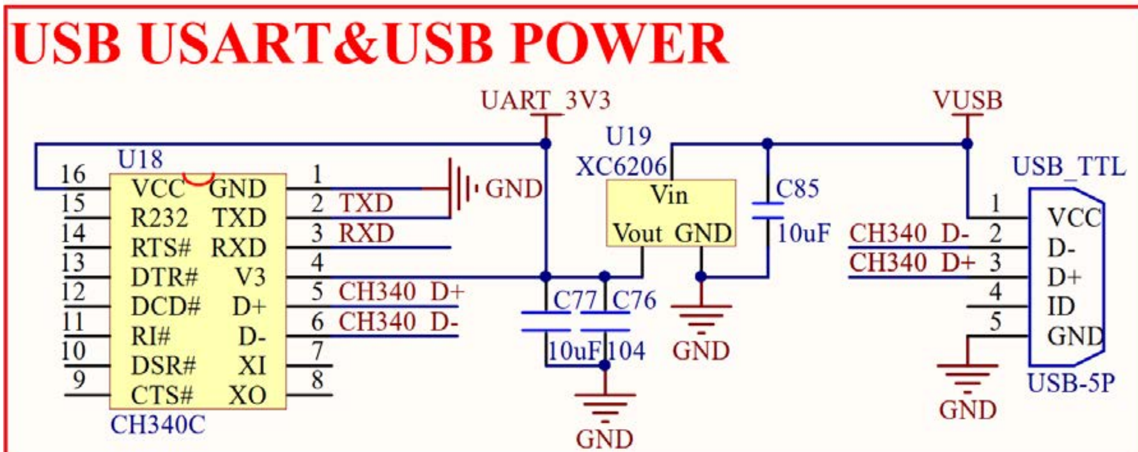


图 5.3.27.1 USB 串口

USB 转串口，我们选择的是 CH340C，是国内芯片公司南京沁恒的产品，稳定性非常不错所以还是多支持下国产。CH340C 内置晶振，因此就不需要再在外面连接一个晶振。图中可以看出 CH340C 的电源为 3.3V，并且是独立供电的，U19 是一个 LDO 芯片，负责给 CH340C 提供 3.3V 的电源。CH340C 的电源不受开发板电源开关控制，只要接上 USB 线 CH340 就会上电。图中 RXD/TXD 接 JP5 的 RXD/TXD，是 CH340 芯片的串口接收和发送脚，可以通过跳线帽连接到 I.MX6U 的串口 1 上。

USB_TTL 是一个 MiniUSB 座，提供 CH340C 和电脑通信的接口，同时可以给开发板和 CH340C 供电，VUSB 就是来自电脑 USB 的电源，USB_TTL 是本开发板的主要供电口。

5.4 I.MX6U 核心板原理图详解

5.4.1 SOC

I.MX6U-ALPHA 开发板配套的 I.MX6U 核心板，采用 MCIMX6Y2CVM05AB(528MHz)或 MCIMX6Y2CVM08AB（800MHz，实际 792MHz）作为主控 CPU，这两款主控都是工业级的。自带 32KB 的 L1 指令和数据 Cache、128KB 的 L2 Cache，集成 NEON，集成双精度硬件浮点计算单元 VFPv3，并具有 128KB OCRAM、2 个通用定时器（GPT）、4 个周期定时器（EPIT）、8 个 PWM、1 个 SDMA 控制器、4 个 ECSPI、3 个看门狗、3 个 SAI、4 个 IIC、7 个串口、2 个 USB（高速，带 PHY）、2 个 FlexCAN、2 个 12 位 ADC、1 个 SPDIF 接口、1 个 SRTC、1 个 RTC、2 个 USDHC 接口、1 个 RGB LCD 控制器（ELCDIF）、2 个 10/100M 以太网 MAC 控制器、1 个摄像头接口、1 个硬件随机数生成器、以及 124 个通用 IO 口等，根据芯片型号的不同主频可以为 528Mhz、700MHz(实际 696MHz)、800MHz(实际 792MHz)，轻松应对各种应用。

SOC 部分的原理图如图 5.4.1.1~图 5.4.1.5（因为原理图比较大，缩小下来可能有点看不清，请大家打开开发板光盘的原理图进行查看）所示：

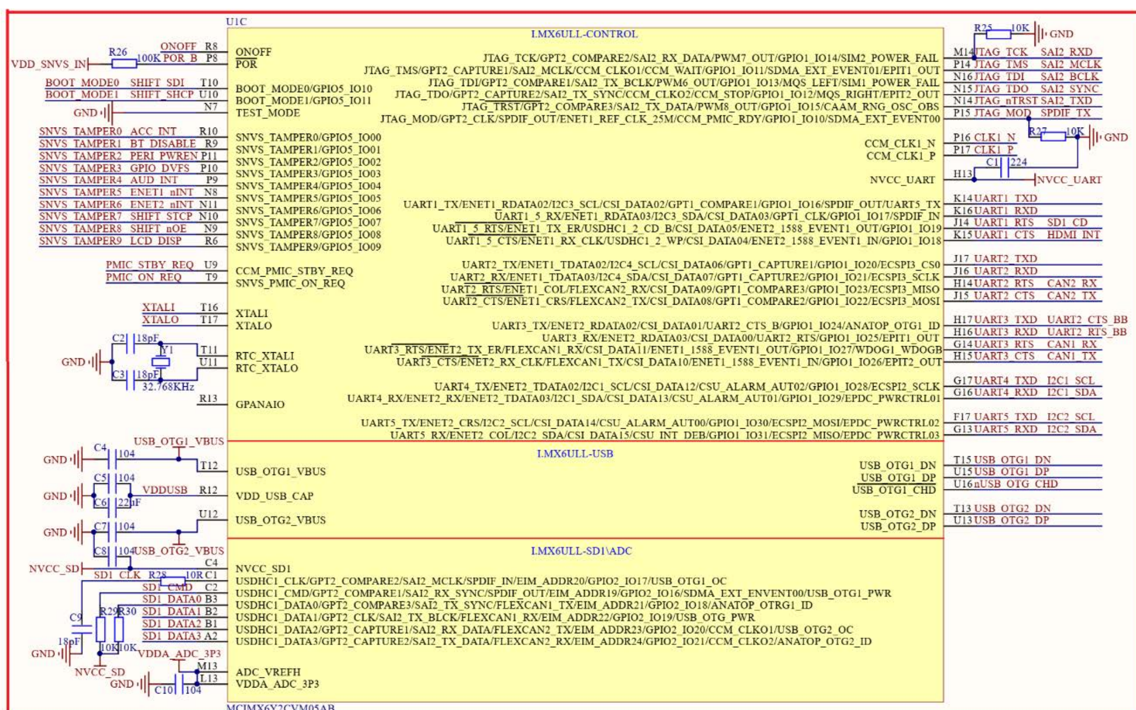


图 5.4.1.1 SOC 部分原理图 1

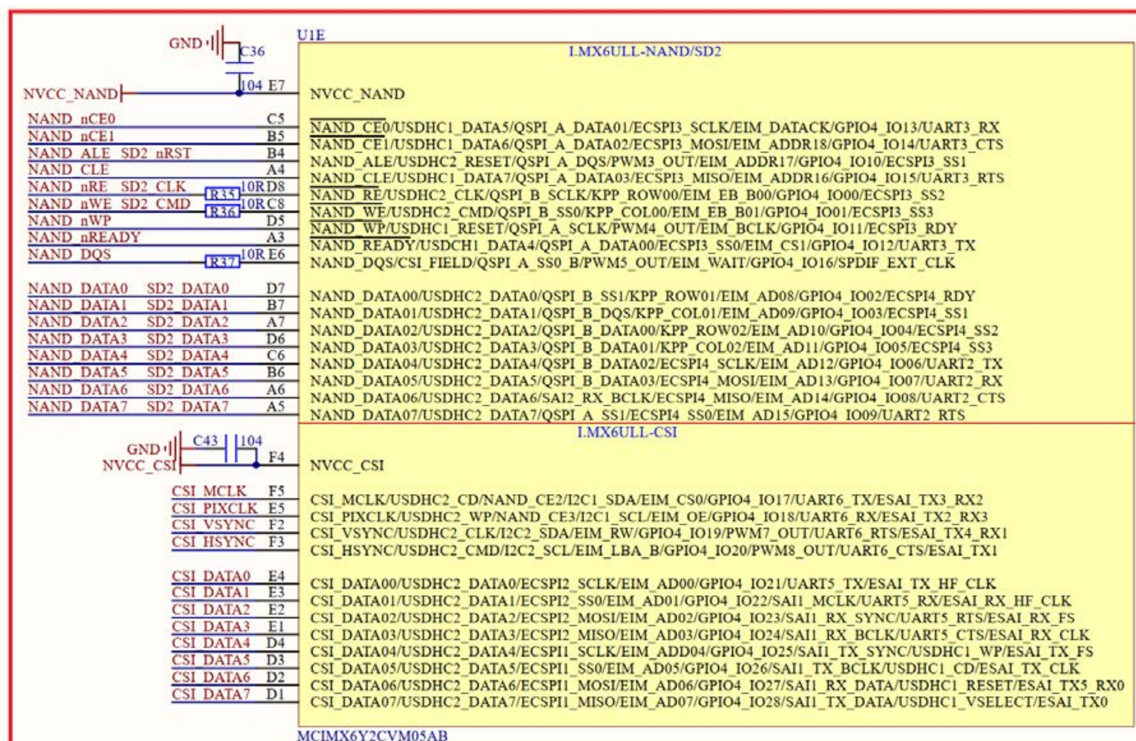


图 5.4.1.2 SOC 部分原理图 2

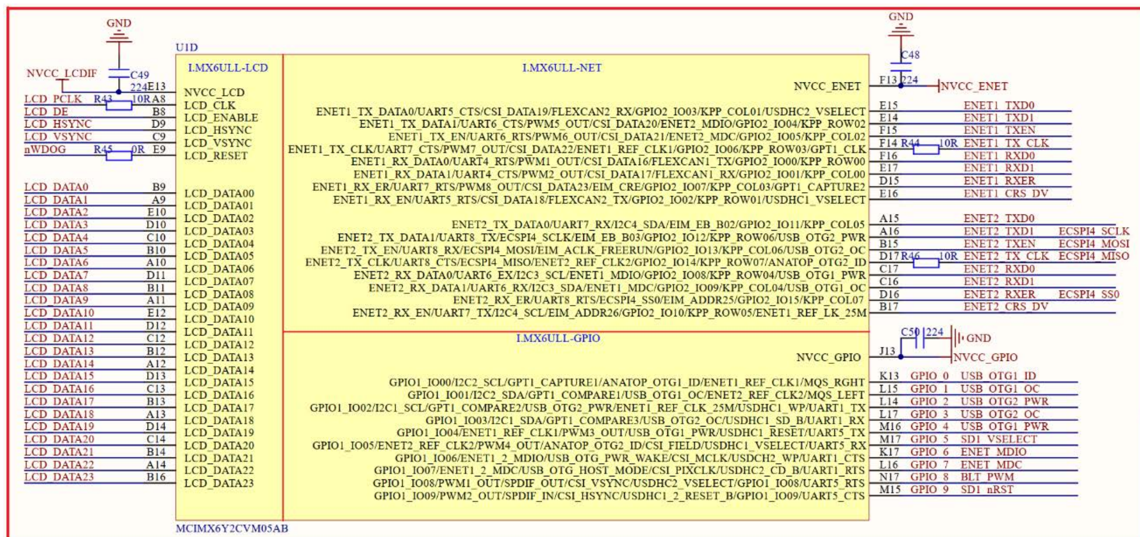


图 5.4.1.3 SOC 部分原理图 3

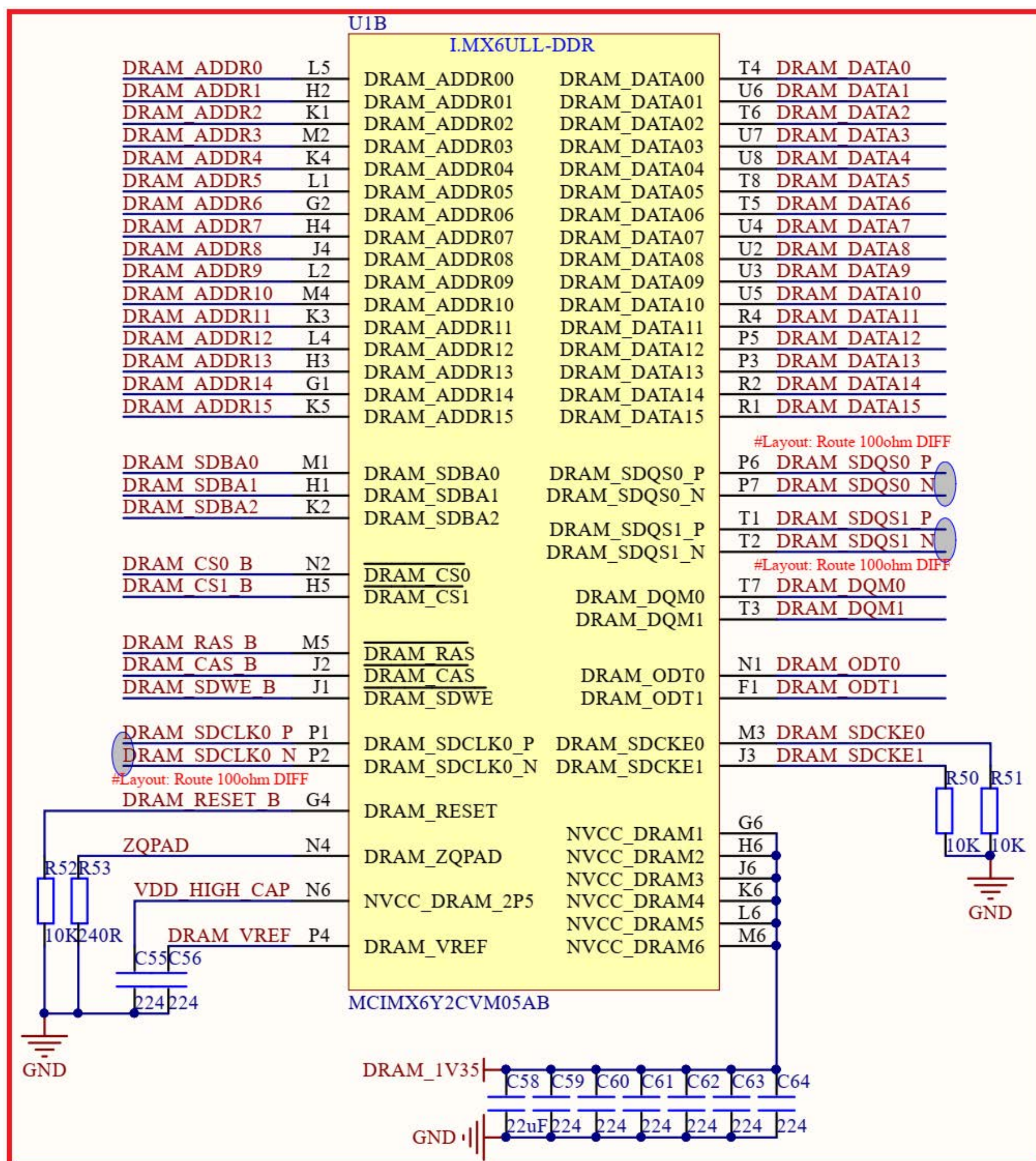


图 5.4.1.4 SOC 部分原理图 4

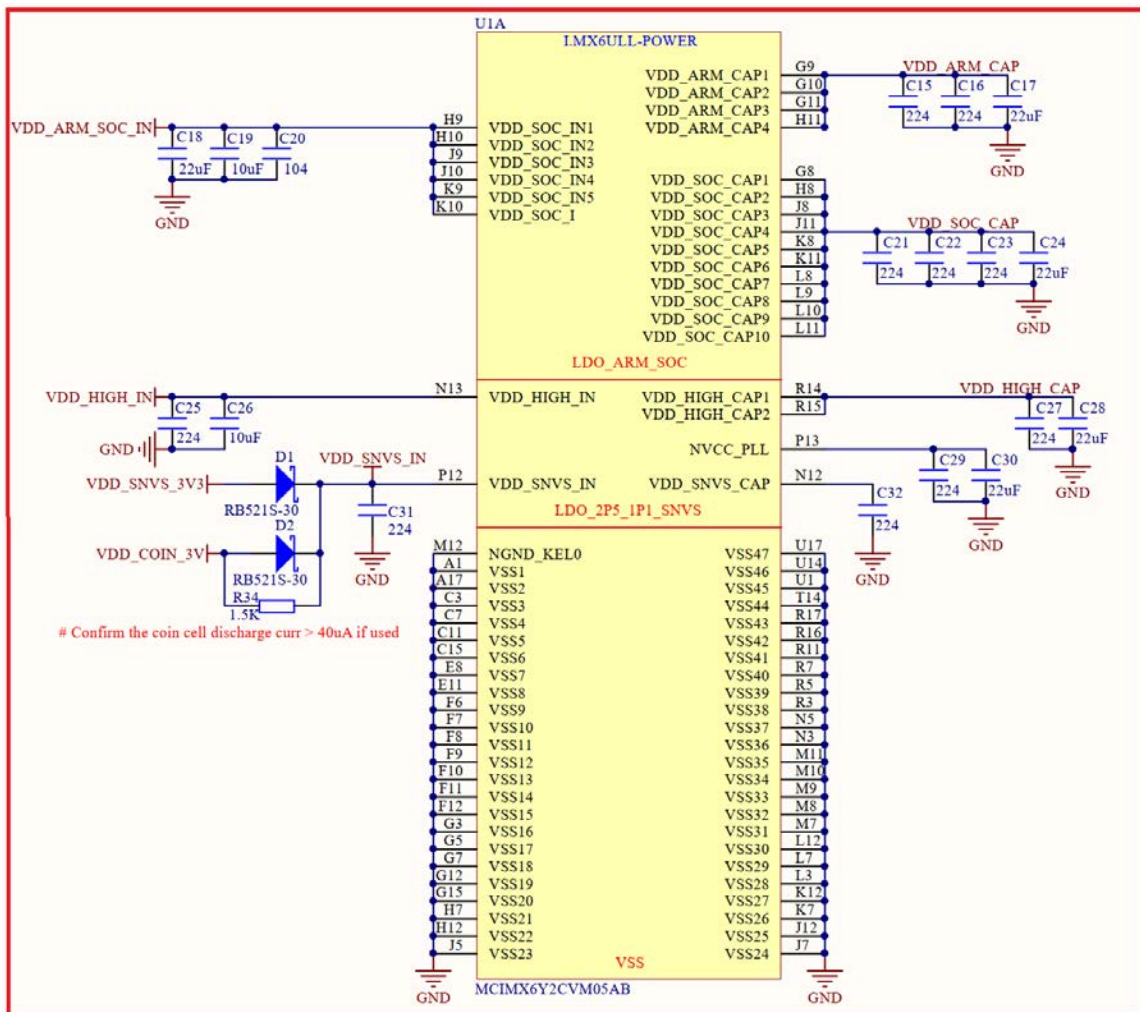


图 5.4.1.5 MCU 部分原理图 5

MCIMX6Y2CVM05AB/08AB 芯片的原理图由 5 个部分组成，接下来依次看一下这五部分的具体内容：

图 5.4.1.1：此部分原理图主要是 I.MX6U 的部分 IO 原理图，比如 SNVS_TAMPER0~9、JTAG 外设 IO、USDHC1 外设 IO、UART 外设 IO、USB 外设 IO 等。

图 5.4.1.2：此部分原理图也是 I.MX6U 的 IO 原理图，主要包括 NAND Flash 外设 IO、USDHC2 外设 IO、CSI 摄像头 IO 等。

图 5.4.1.3：此部分原理图也是 I.MX6U 的 IO 原理图，包括 LCD 外设 IO、ENET 外设 IO、GPIO1_IO01~09 这一组 GPIO。

图 5.4.1.4：此部分原理图是 I.MX6U 的 DRAM 外设 IO。用于连接 DDR 设备，比如正点原子 ALPHA 开发板所使用的 DDR3L。

图 55.4.1.5：此部分原理图是 I.MX6U 的电源部分。

5.4.2 BTB 接口

I.MX6U 核心板采用 2 个 2*30 的 3710M（母座）板对板连接器来同底板连接(在转接板底面)，接插非常方便，转接板上面的底板接口原理图如图 5.4.2.1 所示：

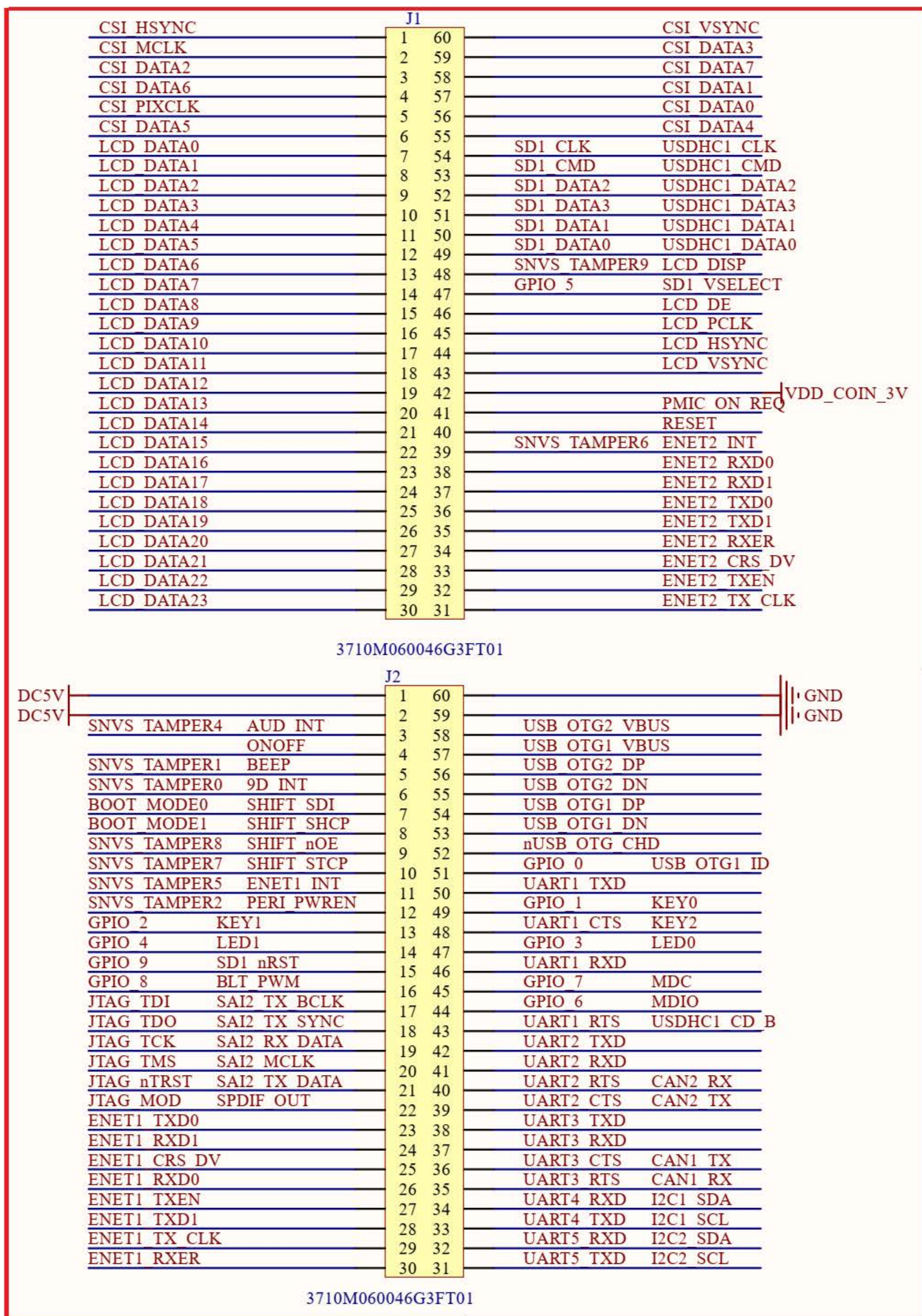


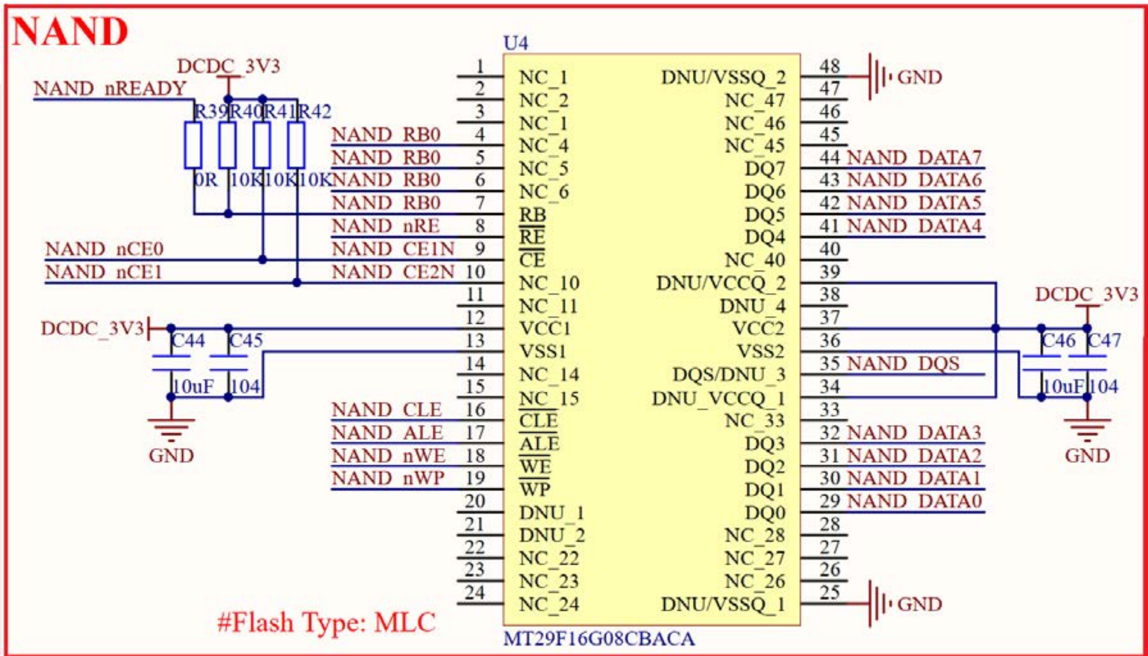
图 5.4.2.1 底板接口

图中，J1 和 J2 是 2 个 2*30 的板对板母座（3710M），和底板的接插非常方便，方便大家嵌入自己的项目中去。该接口总共引出 105 个 IO 口，另外，还有 USB、电源、复位、ONOFF 等

信号。

5.4.3 NAND FLASH

I.MX6U NAND 版本核心板板载了一个 NAND Flash，此部分电路如图 5.4.3.1 所示：

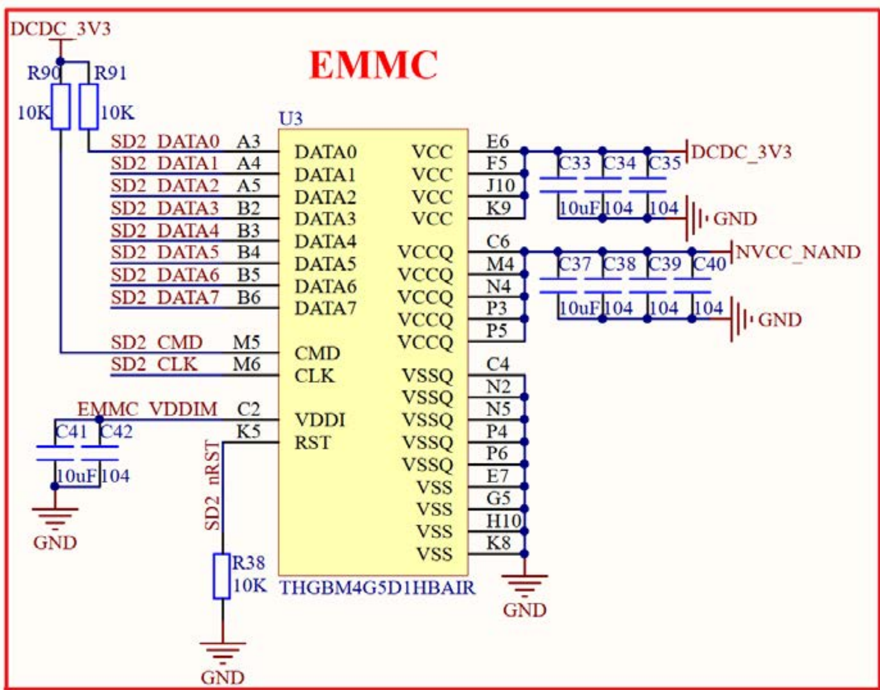


5.4.3.1 NAND Flash

对于 Linux 系统而言，是需要一个存储数据、系统的存储芯片，比如 QSPI Flash、NAND Flash、EMMC 等。正点原子的 I.MX6U-ALPHA 开发板有两种核心板，这两种核心板的 FLASH 存储芯片不同，一个使用的 NAND FLASH、一个使用的 EMMC。图 5.4.3.1 中的是 NAND Flash 的原理图，经过测试，可以支持 256MB、512MB、2GB 的 NAND FLASH 存储芯片。

5.4.4 EMMC

I.MX6U EMMC 核心板板载了 8GB 的 EMMC，此部分电路如图 5.4.4.1 所示：

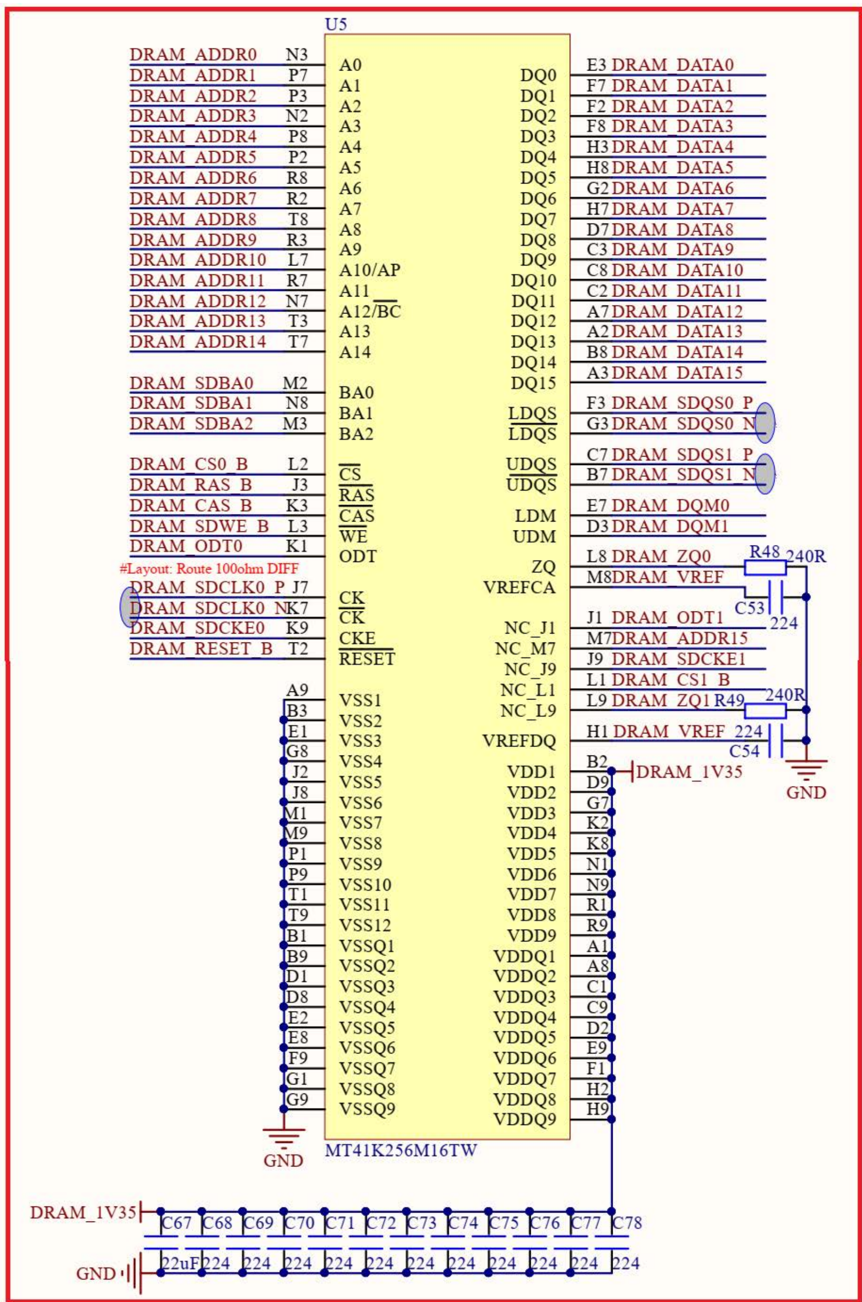


5.4.4.1 EMMC

EMMC 也是存储 Flash，相比 NAND Flash，EMMC 使用简单(和 SD 类似)、速度快、容量高。目前 EMMC 已经逐渐的取代了 NAND Flash，尤其是在手机、平板领域。

5.4.5 DDR3L

I.MX6U 核心板板载了 SDRAM，此部分电路如图 5.4.5.1 所示：



5.4.5.1 DDR3L

图中，U5 就是 DDR3L 芯片，根据配置的不同，一共有两种型号，分别为：NT5CC256M16EP-EK(512MB)和 NT5CC128M16JR-EK(256MB)。该芯片挂在 I.MX6U 的 MMDC 接口上。

5.4.5 核心板电源

I.MX6U 对于供电有严格的要求，尤其是上电顺序，正点原子的 I.MX6U 核心板供电主要分 5 部分：SNVS 供电、DCDC_3V3 供电、ARM/SOC 内核供电、DDR3L 供电和 SD 卡供电，我们依次来看一下，首先是 SNVS 供电，I.MX6U 的数据手册要求，SNVS 必须最先上电，此部分供电电路如图 5.4.5.1 所示：

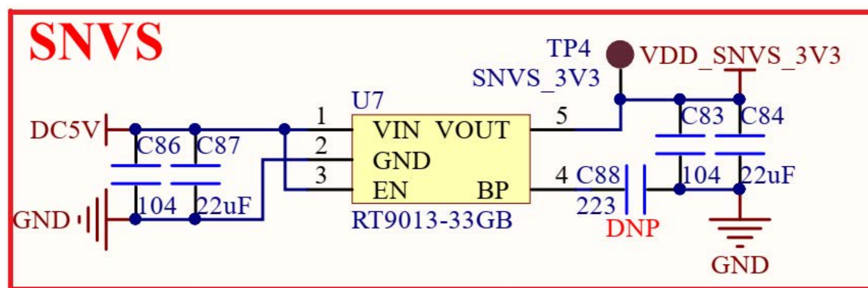


图 5.4.5.1 SVS_3V3 供电

图中，U7 是一颗 LDO 芯片，将 5V 转化为 3.3V，作为 SNVS_3V3，由于 SNVS_3V3 电流不大，所以一个 LDO 芯片就可以了。接下来是 DCDC_3V3，也就是核心板主的电源，如图 5.4.5.2 所示：

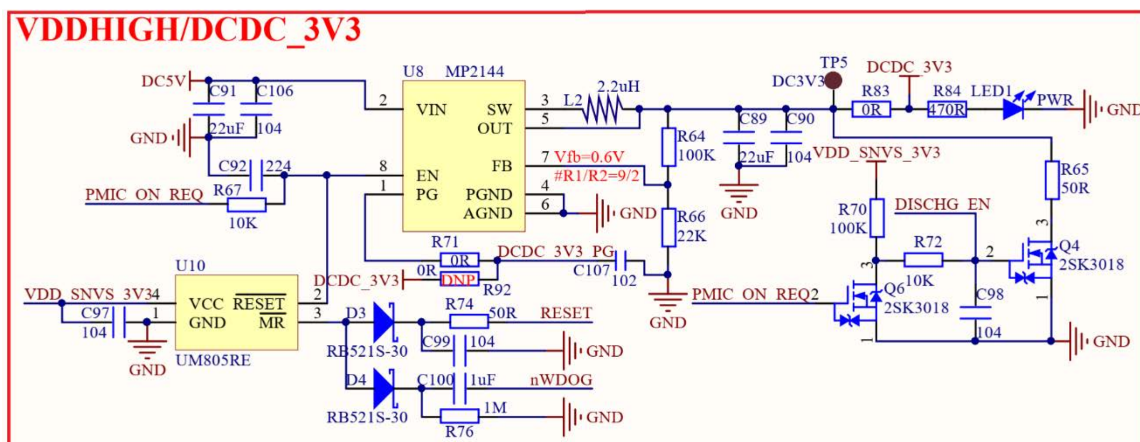


图 5.4.5.2 DCDC_3V3

U8 是一个 DCDC 芯片，用于将 5V 转换为 3.3V，但是电流大，MP2144 最大输出 2A 大电流，因此作为核心板的 3.3V 主电源。这里要注意，U8 的使能引脚使用了 I.MX6U 的 PMIC_ON_REQ 引脚来控制，当 SNVS_3V3 供给 I.MX6U 的 VDD_SNVS_IN 引脚以后，I.MX6U 的 PMIC_ON_REQ 引脚就会输出高电平，从而产生 DCDC_3V3，DCDC_3V3 也是 I.MX6U 的 VDD_HIGH_IN 电源。接下来就是 ARM/SOC 内核电源，如图 5.4.5.3 所示：

图 5.4.5.3 ARM/SOC 电源

U6 也是一片 DCDC，用于产生 ARM/SOC 内核电压，此内核电压可以通过 I.MX6U 的

GPIO_DVFS 和 PMIC_STBY_REQ 来调节。U6 的使能脚连接到了 DCDC_3V3_PG 信号上，此信号是由 U8 产生的，当 U8 输出 3.3V 电压以后 DCDC_3V3_PG 信号就会产生，为一个高电平信号。通过 DCDC_3V3_PG 来控制 VDD_ARM_SOC_IN 电源的产生，这样就保证了 VDD_HIGH_IN 比 ARM_SOC_IN 先上电的要求。接下来看一下 DDR3L 电源，正点原子的 I.MX6U 核心板使用的是 DDR3L，DDDR3L 的工作电压为 1.35V，此部分电源电路如图 5.4.5.4 所示：

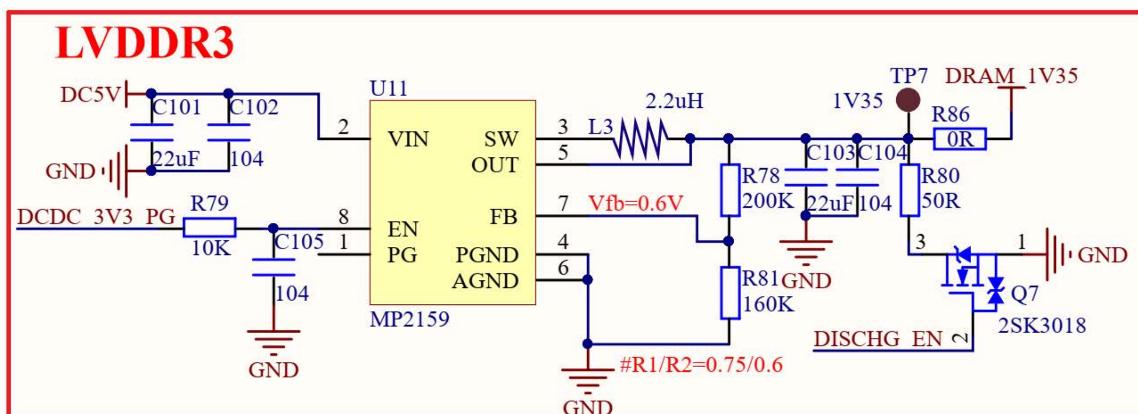


图 5.4.5.4 DDR3L 电路

U11 也是一个 DCDC 芯片，用于将 5V 电源转换为 1.35V 供 DDR3L 使用。接下来看一下 SD 卡部分电路，如图 5.4.5.5 所示：

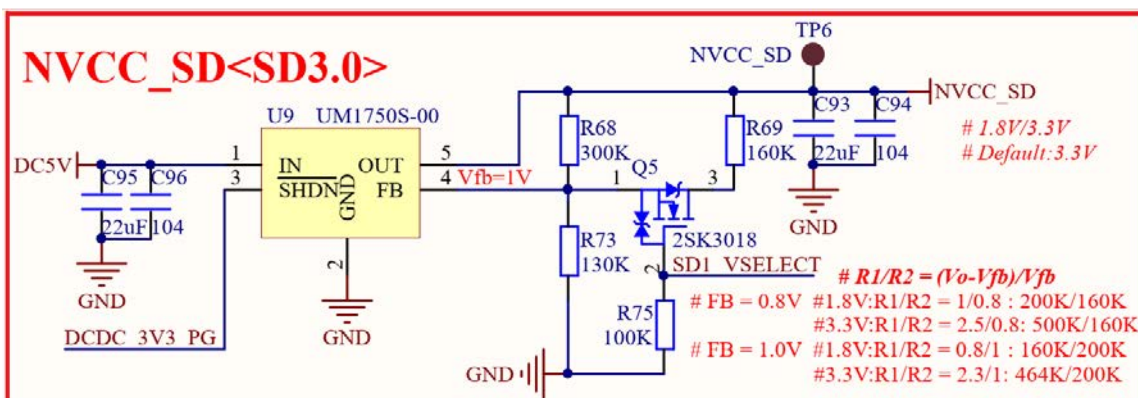


图 5.4.5.5 SD 卡电源

U11 是一片 LDO，用于将 5V 电源转为 3.3V 或 1.8V 供 SD 卡使用，因为高速 SD 卡需要 1.8V 供电，因此此路电源电压是可调的，通过 SD1_VSELECT 来选择使用 3.3V 还是 1.8V，SD1_VSELECT 连接到了 I.MX6U 的 GPIO1_IO05 上。

最后还有 I.MX6U 其他外设电源，如图 5.4.5.6 所示：

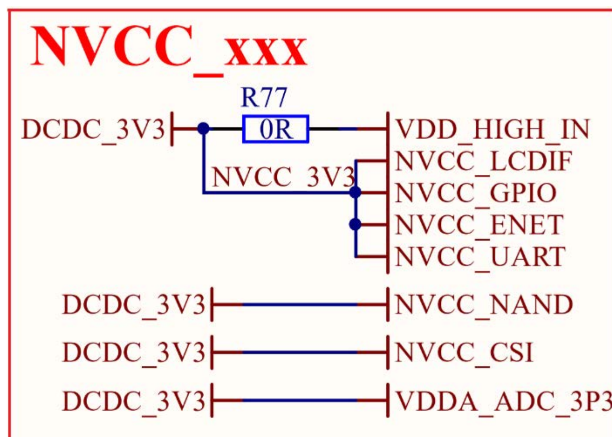


图 5.4.5.6 I.MX6U 其他外设电源

5.5 开发板使用注意事项

为了让大家更好的使用正点原子 I.MX6U-ALPHA 开发板，我们在这里总结该开发板使用的时候尤其要注意的一些问题，希望大家在使用的时候多多注意，以减少不必要的问题。

①、1 个 USB 供电最多 500mA，且由于导线电阻存在，供到开发板的电压，一般都不会有 5V，如果使用了很多大负载外设，比如 4.3 寸屏、网络、摄像头模块等，那么可能引起 USB 供电不够，所以如果是使用 4.3 屏的朋友，或者同时用到多个模块的时候，**建议大家使用一个独立电源供电**。 如果没有独立电源，建议可以同时插 2 个 USB 口，并插上仿真器，这样供电可以更足一些。

②、当你想使用某个 IO 口用作其他用处的时候，请先看看开发板的原理图，该 IO 口是否有连接在开发板的某个外设上，如果有，该外设的这个信号是否会对你的使用造成干扰，先确定无干扰，再使用这个 IO。

③、开发板上的跳线帽比较多，大家在使用某个功能的时候，要先查查这个是否需要设置跳线帽，以免浪费时间。

④、当液晶显示白屏的时候，请先检查液晶模块是否插好（拔下来重新插试试）。

⑤、开发板的 USB OTG 的 USB SLAVE 和 USB HOST 共用同一个 USB 口，所以，他们不可以同时使用。使用的时候多加注意。

⑥、当需要从底板上拆转接板下来的时候，请左右晃动取下，不要太大幅度，否则有可能拆坏座子。

至此，本手册的实验平台（正点原子 I.MX6U-ALPHA 开发板）的硬件部分就介绍完了，了解了整个硬件对我们后面的学习会有很大帮助，有助于理解后面的代码，在编写软件的时候，可以事半功倍，希望大家细读！另外正点原子开发板的其他资料及教程更新，都可以在技术论坛 www.openedv.com 下载到，大家可以经常去这个论坛获取更新的信息。

第六章 Cortex-A7 MPCore 架构

I.MX6UL 使用的是 Cortex-A7 内核，本章就给大家介绍一下 Cortex-A7 架构的一些基本知识。了解了 Cortex-A7 架构以后有利于我们后面的学习，因为后面有很多例程涉及到 Cortex-A7 架构方面的知识，比如处理器模型、Cortex-A7 寄存器组等等，但是 Cortex-A7 架构很庞大，远不是一章就能讲完的，所以本章只是对 Cortex-A7 架构做基本的讲解，主要是为我们后续的试验打基础。

本章参考了《Cortex-A7 Technical ReferenceManua.pdf》和《ARM Cortex-A(armV7)编程手册 V4.0.pdf》这俩份文档，这两份文档都是 ARM 官方的文档，详细的介绍了 Cortex-A7 架构和 ARMv7-A 指令集。这两份文档我们都已经放到了开发板光盘中，路径为：开发板光盘->4、参考资料。

6.1 Cortex-A7 MPCore 简介

Cortex-A7 MPcore 处理器支持 1~4 核，通常是和 Cortex-A15 组成 big.LITTLE 架构的，Cortex-A15 作为大核负责高性能运算，比如玩游戏啥的，Cortex-A7 负责普通应用，因为 Cortex-A7 省电。Cortex-A7 本身性能也不弱，不要看它叫做 Cortex-A7 但是它可是比 Cortex-A8 性能要强大，而且更省电。ARM 官网对于 Cortex-A7 的说明如下：

"在 28nm 工艺下，Cortex-A7 可以运行在 1.2~1.6GHz，并且单核面积不大于 0.45mm^2(含有浮点单元、NEON 和 32KB 的 L1 缓存)，在典型场景下功耗小于 100mW， 这使得它非常适合对功耗要求严格的移动设备，这意味着 Cortex-A7 在获得与 Cortex-A9 相似性能的情况下，其功耗更低"。

Cortex-A7 MPCore 支持在一个处理器上选配 1~4 个内核，Cortex-A7 MPCore 多核配置如图 6.1.1 所示：

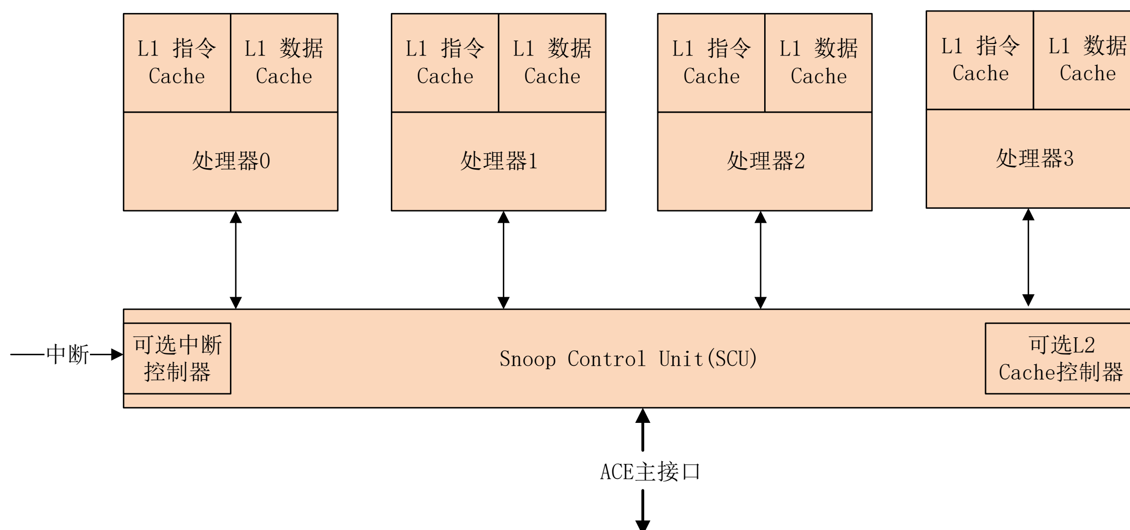


图 6.1.1 多核配置图

Cortex-A7 MPCore 的 L1 可选择 8KB、16KB、32KB、64KB，L2 Cache 可以不配，也可以选择 128KB、256KB、512KB、1024KB。I.MX6UL 配置了 32KB 的 L1 指令 Cache 和 32KB 的 L1 数据 Cache，以及 128KB 的 L2 Cache。Cortex-A7MPCore 使用 ARMv7-A 架构，主要特性如下：

、SIMDv2 扩展整形和浮点向量操作。

、提供了与 ARM VFPv4 体系结构兼容的高性能的单双精度浮点指令，支持全功能的 IEEE754。

、支持大物理扩展(LPAE)，最高可以访问 40 位存储地址，也就是最高可以支持 1TB 的内存。

、支持硬件虚拟化。

、支持 Generic Interrupt Controller(GIC)V2.0。

、支持 NEON，可以加速多媒体和信号处理算法。

6.2 Cortex-A 处理器运行模型

以前的 ARM 处理器有 7 中运行模型：User、FIQ、IRQ、Supervisor(SVC)、Abort、Undef 和 System，其中 User 是非特权模式，其余 6 中都是特权模式。但新的 Cortex-A 架构加入了 TrustZone 安全扩展，所以就新加了一种运行模式：Monitor，新的处理器架构还支持虚拟化扩展，因此又加入了另一个运行模式：Hyp，所以 Cortex-A7 处理器有 9 中处理模式，如表 6.2.1 所示：

模式	描述
User(USR)	用户模式，非特权模式，大部分程序运行的时候就处于此模式。
FIQ	快速中断模式，进入 FIQ 中断异常
IRQ	一般中断模式。
Supervisor(SVC)	超级管理员模式，特权模式，供操作系统使用。
Monitor(MON)	监视模式？这个模式用于安全扩展模式，只用户安全。
Abort(ABT)	数据访问终止模式，用于虚拟存储以及存储保护。
Hyp(HYP)	超级监视模式？用于虚拟化扩展。
Undef(UND)	未定义指令终止模式。
System(SYS)	系统模式，用于运行特权级的操作系统任务

表 6.2.1 九种运行模式

在表 6.2.1.9 中，除了 User(USR)用户模式以外，其它 8 种运行模式都是特权模式，在特权模式下，程序可以访问所有的系统资源。这几个运行模式可以通过软件进行任意切换，也可以通过中断或者异常来进行切换。大多数的程序都运行在用户模式，用户模式下是不能访问系统所有资源的，有些资源是受限的，要想访问这些受限的资源就必须进行模式切换。但是用户模式是不能直接进行切换的，用户模式下需要借助异常来完成模式切换，当要切换模式的时候，应用程序可以产生异常，在异常的处理过程中完成处理器模式切换。

当中断或者异常发生以后，处理器就会进入到相应的异常模式种，每一种模式都有一组寄存器供异常处理程序使用，这样的目的是为了保证在进入异常模式以后，用户模式下的寄存器不会被破坏。

如果学过 STM32 和 UCOS、FreeRTOS 就会知道，STM32 只有两种运行模式，特权模式和非特权模式，但是 Cortex-A 就有 9 种运行模式。

6.3 Cortex-A 寄存器组

本节我们要讲的是 Cortex-A 的内核寄存器组，注意不是芯片的外设寄存器，本节主要参考《ARM Cortex-A(armV7)编程手册 V4.0.pdf》的“第 3 章 ARM Processor Modes And Registers”。

ARM 架构提供了 16 个 32 位的通用寄存器(R0~R15)供软件使用，前 15 个(R0~R14)可以用作通用的数据存储，R15 是程序计数器 PC，用来保存将要执行的指令。ARM 还提供了一个当前程序状态寄存器 CPSR 和一个备份程序状态寄存器 SPSR，SPSR 寄存器就是 CPSR 寄存器的备份。这 18 个寄存器如图 6.3.1 所示：

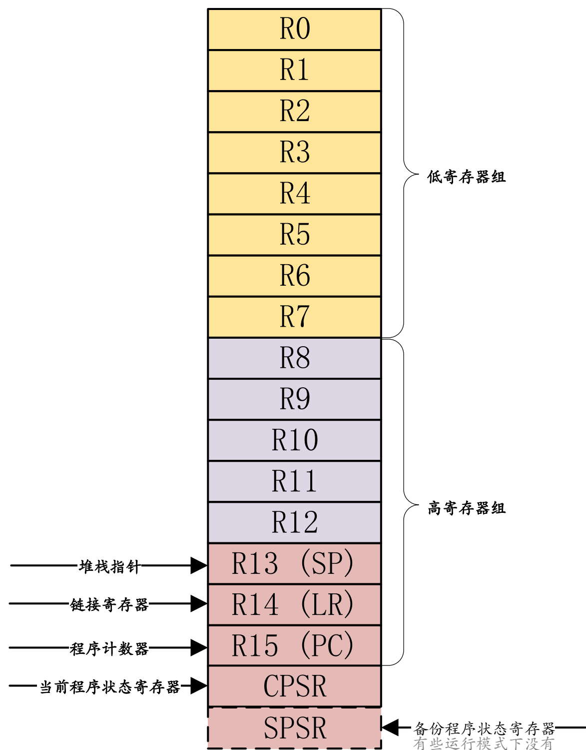


图 6.3.1 Cortex-A 寄存器。

上一小节我们讲了 Cortex-A7 有 9 种运行模式，每一种运行模式都有一组与之对应的寄存器组。每一种模式可见的寄存器包括 15 个通用寄存器(R0~R14)、一两个程序状态寄存器和一个程序计数器 PC。在这些寄存器中，有些是所有模式所共用的同一个物理寄存器，有一些是各模式自己所独立拥有的，各个模式所拥有的寄存器如表 6.3.2 所示：

User	Sys	FIQ	IRQ	ABT	SVC	UND	MON	HYP
R0	R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12	R12	R12
R13(sp)	R13(sp)	SP_fiq	SP_irq	SP_abt	SP_svc	SP_und	SP_mon	SP_hyp
R14(lr)	R14(lr)	LR_fiq	LR_irq	LR_abt	LR_svc	LR_und	LR_mon	R14(lr)
R15(pc)	R15(pc)	R15(pc)	R15(pc)	R15(pc)	R15(pc)	R15(pc)	R15(pc)	R15(pc)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_abt	SPSR_svc	SPSR_und	SPSR_mon	SPSR_hyp
								ELR_hyp

图 6.3.2 九种模式所对应的寄存器

从图 6.3.2 中浅色字体的是与 User 模式所共有的寄存器，蓝绿色背景的是各个模式所独有的寄存器。可以看出，在所有的模式中，低寄存器组(R0~R7)是共享同一组物理寄存器的，只是

一些高寄存器组在不同的模式有自己独有的寄存器，比如 FIQ 模式下 R8~R14 是独立的物理寄存器。假如某个程序在 FIQ 模式下访问 R13 寄存器，那它实际访问的是寄存器 R13_fiq，如果程序处于 SVC 模式下访问 R13 寄存器，那它实际访问的是寄存器 R13_svc。总结一下，Cortex-A 内核寄存器组成如下：

①、34 个通用寄存器，包括 R15 程序计数器(PC)，这些寄存器都是 32 位的。

②、8 个状态寄存器，包括 CPSR 和 SPSR。

③、Hyp 模式下独有一个 ELR_Hyp 寄存器。

6.3.1 通用寄存器

R0~R15 就是通用寄存器，通用寄存器可以分为一下三类：

①、未备份寄存器，即 R0~R7。

②、备份寄存器，即 R8~R14。

③、程序计数器 PC，即 R15。

分别来看一下这三类寄存器：

1、未备份寄存器

未备份寄存器指的是 R0~R7 这 8 个寄存器，因为在所有的处理器模式下这 8 个寄存器都是同一个物理寄存器，在不同的模式下，这 8 个寄存器中的数据就会被破坏。所以这 8 个寄存器并没有被用作特殊用途。

2、备份寄存器

备份寄存器中的 R8~R12 这 5 个寄存器有两种物理寄存器，在快速中断模式下(FIQ)它们对应着 Rx_irq(x=8~12)物理寄存器，其他模式下对应着 Rx(8~12)物理寄存器。FIQ 是快速中断模式，看名字就是知道这个中断模式要求快速执行！ FIQ 模式下中断处理程序可以使用 R8~R12 寄存器，因为 FIQ 模式下的 R8~R12 是独立的，因此中断处理程序可以不用执行保存和恢复中断现场的指令，从而加速中断的执行过程。

备份寄存器 R13 一共有 8 个物理寄存器，其中一个是用户模式(User)和系统模式(Sys)共用的，剩下的 7 个分别对应 7 种不同的模式。R13 也叫做 SP，用来做为栈指针。基本上每种模式都有一个自己的 R13 物理寄存器，应用程序会初始化 R13，使其指向该模式专用的栈地址，这就是常说的初始化 SP 指针。

备份寄存器 R14 一共有 7 个物理寄存器，其中一个是用户模式(User)、系统模式(Sys)和超级监视模式(Hyp)所共有的，剩下的 6 个分别对应 6 种不同的模式。R14 也称为连接寄存器(LR)，LR 寄存器在 ARM 中主要用作如下两种用途：

①、每种处理器模式使用 R14(LR)来存放当前子程序的返回地址，如果使用 BL 或者 BLX 来调用子函数的话，R14(LR)被设置成该子函数的返回地址，在子函数中，将 R14(LR)中的值赋给 R15(PC)即可完成子函数返回，比如在子程序中可以使用如下代码：

```
MOV PC, LR  @寄存器 LR 中的值赋值给 PC，实现跳转
```

或者可以在子函数的入口出将 LR 入栈：

```
PUSH {LR}   @将 LR 寄存器压栈
```

在子函数的最后面出栈即可：

```
POP {PC}    @将上面压栈的 LR 寄存器数据出栈给 PC 寄存器
```

②、当异常发生以后，该异常模式对应的 R14 寄存器被设置成该异常模式将要返回的地址，R14 也可以当作普通寄存器使用。

3、程序计数器 R15

程序计数器 R15 也叫做 PC，R15 保存着当前执行的指令地址值加 8 个字节，这是因为 ARM 的流水线机制导致的。ARM 处理器 3 级流水线：取指->译码->执行，这三级流水线循环执行，比如当前正在执行第一条指令的同时也对第二条指令进行译码，第三条指令也同时被取出存放在 R15(PC)中。我们喜欢以当前正在执行的指令作为参考点，也就是以第一条指令为参考点，那么 R15(PC)中存放的就是第三条指令，换句话说就是 R15(PC)总是指向当前正在执行的指令地址再加上 2 条指令的地址。对于 32 位的 ARM 处理器，每条指令是 4 个字节，所以:

R15 (PC)值 = 当前执行的程序位置 + 8 个字节。

6.3.2 程序状态寄存器

所有的处理器模式都共用一个 CPSR 物理寄存器，因此 CPSR 可以在任何模式下被访问。CPSR 是当前程序状态寄存器，该寄存器包含了条件标志位、中断禁止位、当前处理器模式标志等一些状态位以及一些控制位。所有的处理器模式都共用一个 CPSR 必然会导致冲突，为此，除了 User 和 Sys 这两个模式以外，其他 7 个模式每个都配备了一个专用的物理状态寄存器，叫做 SPSR(备份程序状态寄存器)，当特定的异常中断发生时，SPSR 寄存器用来保存当前程序状态寄存器(CPSR)的值，当异常退出以后可以用 SPSR 中保存的值来恢复 CPSR。

因为 User 和 Sys 这两个模式不是异常模式，所以并没有配备 SPSR，因此不能在 User 和 Sys 模式下访问 SPSR，会导致不可预知的结果。由于 SPSR 是 CPSR 的备份，因此 SPSR 和 CPSR 的寄存器结构相同，如图 6.3.2.1 所示：

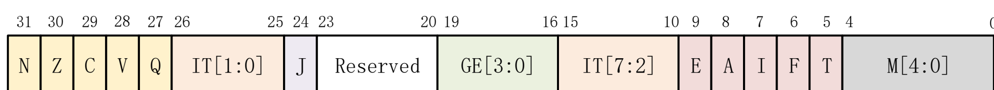


图 6.3.2.1 CPSR 寄存器

N(bit31)：当两个补码表示的 有符号整数运算的时候，N=1 表示运算对的结果为负数，N=0 表示结果为正数。

Z(bit30)：Z=1 表示运算结果为零，Z=0 表示运算结果不为零，对于 CMP 指令，Z=1 表示进行比较的两个数大小相等。

C(bit29)：在加法指令中，当结果产生了进位，则 C=1，表示无符号数运算发生上溢，其它情况下 C=0。在减法指令中，当运算中发生借位，则 C=0，表示无符号数运算发生下溢，其它情况下 C=1。对于包含移位操作的非加/减法运算指令，C 中包含最后一次溢出的位的数值，对于其它非加/减运算指令，C 位的值通常不受影响。

V(bit28)：对于加/减法运算指令，当操作数和运算结果表示为二进制的补码表示的带符号数时，V=1 表示符号位溢出，通常其他位不影响 V 位。

Q(bit27)：仅 ARM v5TE_J 架构支持，表示饱和状态，Q=1 表示累积饱和，Q=0 表示累积不饱和。

IT[1:0](bit26:25)：和 IT[7:2](bit15:bit10)一起组成 IT[7:0]，作为 IF-THEN 指令执行状态。

J(bit24)：仅 ARM_v5TE-J 架构支持，J=1 表示处于 Jazelle 状态，此位通常和 T(bit5)位一起表示当前所使用的指令集，如表 6.3.2.1 所示：

J	T	描述
0	0	ARM
0	1	Thumb
1	1	ThumbEE
1	0	Jazelle

表 6.3.2.1 指令类型

GE[3:0](bit19:16)：SIMD 指令有效，大于或等于。

IT[7:2](bit15:10)：参考 IT[1:0]。

E(bit9)：大小端控制位，E=1 表示大端模式，E=0 表示小端模式。

A(bit8)：禁止异步中断位，A=1 表示禁止异步中断。

I(bit7)：I=1 禁止 IRQ，I=0 使能 IRQ。

F(bit6)：F=1 禁止 FIQ，F=0 使能 FIQ。

T(bit5)：控制指令执行状态，表明本指令是 ARM 指令还是 Thumb 指令，通常和 J(bit24)一起表明指令类型，参考 J(bit24)位。

M[4:0]：处理器模式控制位，含义如表 6.3.2.2 所示：

M[4:0]	处理器模式
10000	User 模式
10001	FIQ 模式
10010	IRQ 模式
10011	Supervisor(SVC)模式
10110	Monitor(MON)模式
10111	Abort(ABT)模式
11010	Hyp(HYP)模式
11011	Undef(UND)模式
11111	System(SYS)模式

表 6.3.2.2 处理器模式位

第七章 ARM 汇编基础

我们在学习 STM32 的时候几乎没有用到过汇编，可能在学习 UCOS、FreeRTOS 等 RTOS 类操作系统移植的时候可能会接触到一点汇编。但是我们在进行嵌入式 Linux 开发的时候是绝对要掌握基本的 ARM 汇编，因为 Cortex-A 芯片一上电 SP 指针还没初始化，C 环境还没准备好，所以肯定不能运行 C 代码，必须先用汇编语言设置好 C 环境，比如初始化 DDR、设置 SP 指针等等，当汇编把 C 环境设置好了以后才可以运行 C 代码。所以 Cortex-A 一开始肯定是汇编代码，其实 STM32 也一样的，一开始也是汇编，以 STM32F103 为例，启动文件 startup_stm32f10x_hd.s 就是汇编文件，只是这个文件 ST 已经写好了，我们根本不用去修改，所以大部分学习者都没有深入的去研究。汇编的知识很庞大，本章我们只讲解最常用的一些指令，满足我们后续学习即可。

I.MX6U-ALPHA 使用的是 NXP 的 I.MX6UL 芯片，这是一款 Cortex-A7 内核的芯片，所以我们主要讲的是 Cortex-A 的汇编指令。为此我们需要参考两份跟 Cortex-A 内核有关的文档：《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》和《ARM Cortex-A(armV7)编程手册 V4.0.pdf》，第一份文档主要讲解 ARMv7-A 和 ARMv7-R 指令集的开发，Cortex-A7 使用的是 ARMv7-A 指令集，第二份文档主要讲解 Cortex-A(armV7)编程的，这两份文档是学习 Cortex-A 不可或缺的文档。在《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》的 A4 章详细的讲解了 Cortex-A 的汇编指令，要想系统的学习 Cortex-A 的指令就要认真的阅读 A4 章节。

对于 Cortex-A 芯片来讲，大部分芯片在上电以后 C 语言环境还没准备好，所以第一行程序肯定是汇编的，至于要写多少汇编程序，那就看你能在哪一步把 C 语言环境准备好。所谓的 C 语言环境就是保证 C 语言能够正常运行。C 语言中的函数调用涉及到出栈入栈，出栈入栈就要对堆栈进行操作，所谓的堆栈其实就是一段内存，这段内存比较特殊，由 SP 指针访问，SP 指针指向栈顶。芯片一上电 SP 指针还没有初始化，所以 C 语言没法运行，对于有些芯片还需要初始化 DDR，因为芯片本身没有 RAM，或者内部 RAM 不开放给用户使用，用户代码需要在 DDR 中运行，因此一开始要用汇编来初始化 DDR 控制器。后面学习 Uboot 和 Linux 内核的时候汇编是必须要会的，是不是觉得好难啊？还要会汇编！前面都说了只是在芯片上电以后用汇编来初始化一些外设，不会涉及到复杂的代码，而且使用到的指令都是很简单的，用到的就那么十几个指令。所以，不要看到汇编就觉得复杂，打击学习信心。

7.1 GNU 汇编语法

如果大家使用过 STM32 的话就会知道 MDK 和 IAR 下的启动文件 startup_stm32f10x_hd.s 其中的汇编语法是有所不同的，将 MDK 下的汇编文件直接复制到 IAR 下去编译就会出错，因为 MDK 和 IAR 的编译器不同，因此对于汇编的语法就有一些小区别。我们要编写的是 ARM 汇编，编译使用的 GCC 交叉编译器，所以我们的汇编代码要符合 GNU 语法。

GNU 汇编语法适用于所有的架构，并不是 ARM 独享的，GNU 汇编由一系列的语句组成，每行一条语句，每条语句有三个可选部分，如下：

```
label：instruction @ comment
```

label 即标号，表示地址位置，有些指令前面可能会有标号，这样就可以通过这个标号得到指令的地址，标号也可以用来表示数据地址。注意 label 后面的冒号“：”，任何以冒号“：”结尾的标识符都会被认识是一个标号。

instruction 即指令，也就是汇编指令或伪指令。

@符号，表示后面的是注释，就跟 C 语言里面的“/*”和“*/”一样，其实在 GNU 汇编文件中我们也可以使用“/*”和“*/”来注释。

comment 就是注释内容。

比如如下代码：

```
add:
    MOVS R0, #0X12  @设置 R0=0X12
```

上面代码中“add:”就是标号，“MOVS R0,#0X12”就是指令，最后的“@设置 R0=0X12”就是注释。

注意！ARM 中的指令、伪指令、伪操作、寄存器名等可以全部使用大写，也可以全部使用小写，但是不能大小写混用。

用户可以使用.section 伪操作来定义一个段，汇编系统预定义了一些段名：

.text 表示代码段。

.data 初始化的数据段。

.bss 未初始化的数据段。

.rodata 只读数据段。

我们当然可以自己使用.section 来定义一个段，每个段以段名开始，以下一段名或者文件结尾结束，比如：

```
.section .testsection @定义一个 testsetcion 段
```

汇编程序的默认入口标号是_start，不过我们也可以在链接脚本中使用 ENTRY 来指明其它的入口点，下面的代码就是使用_start 作为入口标号：

```
.global _start

_start:
    ldr r0, =0x12   @r0=0x12
```

上面代码中.global 是伪操作，表示_start 是一个全局标号，类似 C 语言里面的全局变量一样，常见的伪操作有：

.byte 定义单字节数据，比如.byte 0x12。

.short 定义双字节数据，比如.byte 0x1234。

.long 定义一个 4 字节数据，比如.long 0x12345678。

.equ 赋值语句，格式为：.equ 变量名，表达式，比如.equ num, 0x12，表示 num=0x12。

.align 数据字节对齐，比如：.align 4 表示 4 字节对齐。

.end 表示源文件结束。

.global 定义一个全局符号，格式为：.global symbol，比如：.global _start。

GNU 汇编还有其它的伪操作，但是最常见的就是上面这些，如果想详细的了解全部的伪操作，可以参考《ARM Cortex-A(armV7)编程手册 V4.0.pdf》的 57 页。

GNU 汇编同样也支持函数，函数格式如下：

```
函数名:
    函数体
    返回语句
```

GNU 汇编函数返回语句不是必须的，如下代码就是用汇编写的 Cortex-A7 中断服务函数：

示例代码 7.1.1.1 汇编函数定义

```
/* 未定义中断 */
Undefined_Handler:
    ldr r0, =Undefined_Handler
    bx r0

/* SVC 中断 */
SVC_Handler:
    ldr r0, =SVC_Handler
    bx r0

/* 预取终止中断 */
PrefAbort_Handler:
    ldr r0, =PrefAbort_Handler
    bx r0
```

上述代码中定义了三个汇编函数：Undefined_Handler、SVC_Handler 和 PrefAbort_Handler。以函数 Undefined_Handler 为例我们来看一下汇编函数组成，"Undefined_Handler"就是函数名，"ldr r0, =Undefined_Handler"是函数体，"bx r0"是函数返回语句，"bx"指令是返回指令，函数返回语句不是必须的。

7.2 Cortex-A7 常用汇编指令

本节我们将介绍一些常用的 Cortex-A7 汇编指令，如果想系统的了解 Cortex-A7 的所有汇编指令请参考《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》的 A4 章节。

7.2.1 处理器内部数据传输指令

使用处理器做的最多事情就是在处理器内部来回的传递数据，常见的操作有：

①、将数据从一个寄存器传递到另外一个寄存器。

②、将数据从一个寄存器传递到特殊寄存器，如 CPSR 和 SPSR 寄存器。

③、将立即数传递到寄存器。

数据传输常用的指令有三个：MOV、MRS 和 MSR，这三个指令的用法如表 7.2.1.1 所示：

指令	目的	源	描述
MOV	R0	R1	将 R1 里面的数据复制到 R0 中。
MRS	R0	CPSR	将特殊寄存器 CPSR 里面的数据复制到 R0 中。
MSR	CPSR	R1	将 R1 里面的数据复制到特殊寄存器 CPSR 里中。

表 7.2.1.1 常用数据传输指令

分别来详细的介绍一下如何使用这三个指令：

1、MOV 指令

MOV 指令用于将数据从一个寄存器拷贝到另外一个寄存器，或者将一个立即数传递到寄存器里面，使用示例如下：

```
MOV R0，R1           @将寄存器 R1 中的数据传递给 R0，即 R0=R1
MOV R0,  #0X12       @将立即数 0X12 传递给 R0 寄存器，即 R0=0X12
```

2、MRS 指令

MRS 指令用于将特殊寄存器(如 CPSR 和 SPSR)中的数据传递给通用寄存器，要读取特殊寄存器的数据只能使用 MRS 指令！使用示例如下：

```
MRS R0, CPSR     @将特殊寄存器 CPSR 里面的数据传递给 R0，即 R0=CPSR
```

3、MSR 指令

MSR 指令和 MRS 刚好相反，MSR 指令用来将普通寄存器的数据传递给特殊寄存器，也就是写特殊寄存器，写特殊寄存器只能使用 MSR，使用示例如下：

```
MSR CPSR, R0     @将 R0 中的数据复制到 CPSR 中，即 CPSR=R0
```

7.2.2 存储器访问指令

ARM 不能直接访问存储器，比如 RAM 中的数据，I.MX6UL 中的寄存器就是 RAM 类型的，我们用汇编来配置 I.MX6UL 寄存器的时候需要借助存储器访问指令，一般先将要配置的值写入到 Rx(x=0~12)寄存器中，然后借助存储器访问指令将 Rx 中的数据写入到 I.MX6UL 寄存器

中。读取 I.MX6UL 寄存器也是一样的，只是过程相反。常用的存储器访问指令有两种：LDR 和 STR，用法如表 7.2.1.2 所示：

指令	描述
LDR Rd, [Rn , #offset]	从存储器 Rn+offset 的位置读取数据存放到 Rd 中。
STR Rd, [Rn, #offset]	将 Rd 中的数据写入到存储器中的 Rn+offset 位置。

表 7.2.1.2 存储器访问指令

分别来详细的介绍一下如何使用这两个指令：

1、LDR 指令

LDR 主要用于从存储加载数据到寄存器 Rx 中，LDR 也可以将一个立即数加载到寄存器 Rx 中，LDR 加载立即数的时候要使用“=”，而不是“#”。在嵌入式开发中，LDR 最常用的就是读取 CPU 的寄存器值，比如 I.MX6UL 有个寄存器 GPIO1_GDIR，其地址为 0X0209C004，我们现在要读取这个寄存器中的数据，示例代码如下：

示例代码 7.2.2.1 LDR 指令使用

```
1 LDR R0, =0X0209C004  @将寄存器地址 0X0209C004 加载到 R0 中，即 R0=0X0209C004
2 LDR R1, [R0]         @读取地址 0X0209C004 中的数据到 R1 寄存器中
```

上述代码就是读取寄存器 GPIO1_GDIR 中的值，读取到的寄存器值保存在 R1 寄存器中，上面代码中 offset 是 0，也就是没有用到 offset。

2、STR 指令

LDR 是从存储器读取数据，STR 就是将数据写入到存储器中，同样以 I.MX6UL 寄存器 GPIO1_GDIR 为例，现在我们要配置寄存器 GPIO1_GDIR 的值为 0X2000002，示例代码如下：

示例代码 7.2.2.2 STR 指令使用

```
1 LDR R0, =0X0209C004  @将寄存器地址 0X0209C004 加载到 R0 中，即 R0=0X0209C004
2 LDR R1, =0X20000002  @R1 保存要写入到寄存器的值，即 R1=0X20000002
3 STR R1, [R0]         @将 R1 中的值写入到 R0 中所保存的地址中
```

LDR 和 STR 都是按照字进行读取和写入的，也就是操作的 32 位数据，如果要按照字节、半字进行操作的话可以在指令“LDR”后面加上 B 或 H，比如按字节操作的指令就是 LDRB 和 STRB，按半字操作的指令就是 LDRH 和 STRH。

7.2.3 压栈和出栈指令

我们通常会在 A 函数中调用 B 函数，当 B 函数执行完以后再回到 A 函数继续执行。要想在跳回 A 函数以后代码能够接着正常运行，那就必须在跳到 B 函数之前将当前处理器状态保存起来(就是保存 R0~R15 这些寄存器值)，当 B 函数执行完成以后再用前面保存的寄存器值恢复 R0~R15 即可。保存 R0~R15 寄存器的操作就叫做现场保护，恢复 R0~R15 寄存器的操作就叫做恢复现场。在进行现场保护的时候需要进行压栈(入栈)操作，恢复现场就要进行出栈操作。压栈的指令为 PUSH，出栈的指令为 POP，PUSH 和 POP 是一种多存储和多加载指令，即可以一次操作多个寄存器数据，他们利用当前的栈指针 SP 来生成地址，PUSH 和 POP 的用法如表 7.2.3.1 所示：

指令	描述
PUSH <reg list>	将寄存器列表存入栈中。
POP <reg list>	从栈中恢复寄存器列表。

表 7.2.3.1 压栈和出栈指令

假如我们现在要将 R0~R3 和 R12 这 5 个寄存器压栈，当前的 SP 指针指向 0X80000000，

处理器的堆栈是向下增长的，使用的汇编代码如下：

```
PUSH {R0~R3, R12}   @将 R0~R3 和 R12 压栈
```

压栈完成以后的堆栈如图 7.2.3.1 所示：

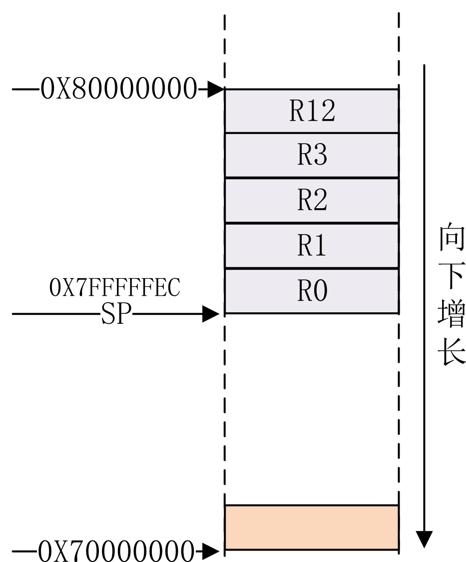


图 7.2.3.1 压栈以后的堆栈

图 7.2.3.1 就是对 R0~R3,R12 进行压栈以后的堆栈示意图，此时的 SP 指向了 0X7FFFFFEC，假如我们现在要再将 LR 进行压栈，汇编代码如下：

```
PUSH {LR}   @将 LR 进行压栈
```

对 LR 进行压栈完成以后的堆栈模型如图 7.2.3.2 所示：

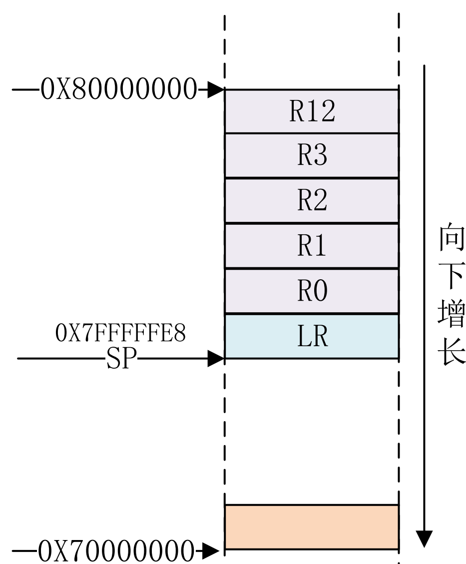


图 7.2.3.2 LR 压栈以后的堆栈

图 7.2.3.2 就是分两步对 R0~R3,R2 和 LR 进行压栈以后的堆栈模型，如果我们要出栈的话就是使用如下代码：

```
POP {LR}          @先恢复 LR
POP {R0~R3,R12} @在恢复 R0~R3,R12
```

出栈的就是从栈顶，也就是 SP 当前执行的位置开始，地址依次减小来提取堆栈中的数据到要恢复的寄存器列表中。PUSH 和 POP 的另外一种写法是“STMFD SP !”和“LDMFD SP!”，

因此上面的汇编代码可以改为：

示例代码 7.2.3.1 STMFD 和 LDMFD 指令
```
1 STMFD SP!,{R0~R3, R12}    @R0~R3,R12 入栈
2 STMFD SP!,{LR}            @LR 入栈
3
4 LDMFD SP!, {LR}           @先恢复 LR
5 LDMFD SP!, {R0~R3, R12}   @再恢复 R0~R3, R12
```

STMFD 可以分为两部分：STM 和 FD，同理，LDMFD 也可以分为 LDM 和 FD。看到 STM 和 LDM 有没有觉得似曾相识(不是 STM32 啊啊啊啊)，前面我们讲了 LDR 和 STR，这两个是数据加载和存储指令，但是每次只能读写存储器中的一个数据。STM 和 LDM 就是多加载和多存储，可以连续的读写存储器中的多个连续数据。

FD 是 Full Descending 的缩写，即满递减的意思。根据 ATPCS 规则,ARM 使用的 FD 类型的堆栈，SP 指向最后最后一个入栈的数值，堆栈是由高地址向下增长的，也就是前面说的向下增长的堆栈，因此最常用的指令就是 STMFD 和 LDMFD。STM 和 LDM 的指令寄存器列表中编号小的对应低地址，编号高的对应高地址。

7.2.4 跳转指令

有多种跳转操作，比如：

①、直接使用跳转指令 B、BL、BX 等。

②、直接向 PC 寄存器里面写入数据。

上述两种方法都可以完成跳转操作，但是一般常用的还是 B、BL 或 BX，用法如表 7.2.4.1：

指令	描述
B <label>	跳转到 label，如果跳转范围超过了+/-2KB，可以指定 B.W <label>使用 32 位版本的跳转指令， 这样可以得到较大范围的跳转
BX <Rm>	间接跳转，跳转到存放于 Rm 中的地址处，并且切换指令集
BL <label>	跳转到标号地址，并将返回地址保存在 LR 中。
BLX <Rm>	结合 BX 和 BL 的特点，跳转到 Rm 指定的地址，并将返回地址保存在 LR 中，切换指令集。

表 7.2.4.1 跳转指令

我们重点来看一下 B 和 BL 指令，因为这两个是我们用的最多的，如果要在汇编中进行函数调用使用的就是 B 和 BL 指令：

1、B 指令

这是最简单的跳转指令，B 指令会将 PC 寄存器的值设置为跳转目标地址， 一旦执行 B 指令，ARM 处理器就会立即跳转到指定的目标地址。如果要调用的函数不会再返回到原来的执行处，那就可以用 B 指令，如下示例：

示例代码 7.2.4.1 B 指令示例
```
1 _start:
2
3   ldr sp,=0X80200000    @设置栈指针
4   b main                @跳转到 main 函数
```

上述代码就是典型的在汇编中初始化 C 运行环境，然后跳转到 C 文件的 main 函数中运行，

上述代码只是初始化了 SP 指针，有些处理器还需要做其他的初始化，比如初始化 DDR 等等。因为跳转到 C 文件以后再也不会回到汇编了，所以在第 4 行使用了 B 指令来完成跳转。

2、BL 指令

BL 指令相比 B 指令，在跳转之前会在寄存器 LR(R14)中保存当前 PC 寄存器值，所以可以通过将 LR 寄存器中的值重新加载到 PC 中来继续从跳转之前的代码处运行，这是子程序调用一个基本但常用的手段。比如 Cortex-A 处理器的 irq 中断服务函数都是汇编写的，主要用汇编来实现现场的保护和恢复、获取中断号等。但是具体的中断处理过程都是 C 函数，所以就会存在汇编中调用 C 函数的问题。而且当 C 语言版本的中断处理函数执行完成以后是需要返回到 irq 汇编中断服务函数，因为还要处理其他的工作，一般是恢复现场。这个时候就不能直接使用 B 指令了，因为 B 指令一旦跳转就再也不会回来了，这个时候要使用 BL 指令，示例代码如下：

示例代码 7.2.4.2 BL 指令示例

```
1 push {r0, r1}              @保存 r0,r1
2 cps #0x13                  @进入 SVC 模式，允许其他中断再次进去
3
5 bl system_irqhandler       @加载 C 语言中断处理函数到 r2 寄存器中
6
7 cps #0x12                  @进入 IRQ 模式
8 pop {r0, r1}
9 str r0, [r1, #0X10]        @中断执行完成，写 EOIR
```

上述代码中第 5 行就是执行 C 语言版的中断处理函数，当处理完成以后是需要返回来继续执行下面的程序，所以使用了 BL 指令。

7.2.5 算术运算指令

汇编中也可以进行算术运算， 比如加减乘除，常用的运算指令用法如表 7.2.5.1 所示：

指令	计算公式	备注
ADD Rd, Rn, Rm	Rd = Rn + Rm	加法运算，指令为 ADD
ADD Rd, Rn, #immed	Rd = Rn + #immed	
ADC Rd, Rn, Rm	Rd = Rn + Rm + 进位	带进位的加法运算，指令为 ADC
ADC Rd, Rn, #immed	Rd = Rn + #immed +进位	
SUB Rd, Rn, Rm	Rd = Rn – Rm	减法
SUB Rd, #immed	Rd = Rd - #immed	
SUB Rd, Rn, #immed	Rd = Rn - #immed	
SBC Rd, Rn, #immed	Rd = Rn - #immed – 借位	带借位的减法
SBC Rd, Rn ,Rm	Rd = Rn – Rm – 借位	
MUL Rd, Rn, Rm	Rd = Rn * Rm	乘法(32 位)
UDIV Rd, Rn, Rm	Rd = Rn / Rm	无符号除法
SDIV Rd, Rn, Rm	Rd = Rn / Rm	有符号除法

表 7.2.5.1 常用运算指令

在嵌入式开发中最常会用的就是加减指令，乘除基本用不到。

7.2.6 逻辑运算指令

我们用 C 语言进行 CPU 寄存器配置的时候常常需要用到逻辑运算符号，比如“&”、“|”等

逻辑运算符。使用汇编语言的时候也可以使用逻辑运算指令，常用的运算指令用法如表 7.2.6.1 所示：

<table>
<tr><th>指令</th><th>计算公式</th><th>备注</th></tr>
<tr><td>AND Rd, Rn</td><td>Rd = Rd &Rn</td><td rowspan="3">按位与</td></tr>
<tr><td>AND Rd, Rn, #immed</td><td>Rd = Rn &#immed</td></tr>
<tr><td>AND Rd, Rn, Rm</td><td>Rd = Rn & Rm</td></tr>
<tr><td>ORR Rd, Rn</td><td>Rd = Rd | Rn</td><td rowspan="3">按位或</td></tr>
<tr><td>ORR Rd, Rn, #immed</td><td>Rd = Rn | #immed</td></tr>
<tr><td>ORR Rd, Rn, Rm</td><td>Rd = Rn | Rm</td></tr>
<tr><td>BIC Rd, Rn</td><td>Rd = Rd & (~Rn)</td><td rowspan="3">位清除</td></tr>
<tr><td>BIC Rd, Rn, #immed</td><td>Rd = Rn & (~#immed)</td></tr>
<tr><td>BIC Rd, Rn , Rm</td><td>Rd = Rn & (~Rm)</td></tr>
<tr><td>ORN Rd, Rn, #immed</td><td>Rd = Rn | (w#immed)</td><td rowspan="2">按位或非</td></tr>
<tr><td>ORN Rd, Rn, Rm</td><td>Rd = Rn | (wRm)</td></tr>
<tr><td>EOR Rd, Rn</td><td>Rd = Rd ^ Rn</td><td rowspan="3">按位异或</td></tr>
<tr><td>EOR Rd, Rn, #immed</td><td>Rd = Rn ^ #immed</td></tr>
<tr><td>EOR Rd, Rn, Rm</td><td>Rd = Rn ^ Rm</td></tr>
</table>

表 7.2.6.1 逻辑运算指令

逻辑运算指令都很好理解，后面时候汇编配置 I.MX6UL 的外设寄存器的时候可能会用到，ARM 汇编就讲解到这里，本节主要讲解了一些最常用的指令，还有很多不常用的指令没有讲解，但是够我们后续学习用了。要想详细的学习 ARM 的所有指令请参考《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》和《ARM Cortex-A(armV7)编程手册 V4.0.pdf》这两份文档。

第八章 汇编 LED 灯试验

本章开始编写本教程第一个裸机例程——经典的点灯试验，这也是我们嵌入式 Linux 学习的第一步。本章使用汇编语言来编写，通过本章了解如何使用汇编语言来初始化 I.MX6U 外设寄存器、了解 I.MX6UL 最基本的 IO 输出功能。万里长征第一步，祝愿大家学习愉快！

8.1 I.MX6U GPIO 详解

8.1.1 STM32 GPIO 回顾

我们一般拿到一款全新的芯片，第一个要做的事情的就是驱动其 GPIO，控制其 GPIO 输出高低电平，我们学习 I.MX6U 也一样的，先来学习一下 I.MX6U 的 GPIO。在学习 I.MX6U 的 GPIO 之前，我们先来回顾一下 STM32 的 GPIO 初始化(如果没有学过 STM32 就不用回顾了)，我们以最常见的 STM32F103 为例来看一下 STM32 的 GPIO 初始化，示例代码如下：

示例代码 8.1.1.1 STM32 GPIO 初始化

```
1  void LED_Init(void)
2  {
3   GPIO_InitTypeDef  GPIO_InitStructure;
4
5   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);//使能 PB 端口时钟
6
7   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;           //PB5 端口配置
8   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    //推挽输出
9   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;   //IO 口速度
10  GPIO_Init(GPIOB, &GPIO_InitStructure); //根据设定参数初始化 GPIOB.5
11
12  GPIO_SetBits(GPIOB,GPIO_Pin_5);        //PB.5 输出高
13 }
```

上述代码就是使用库函数来初始化 STM32 的一个 IO 为输出功能，可以看出上述初始化代码中重点要做的事情有一下几个：

①、使能指定 GPIO 的时钟。

②、初始化 GPIO，比如输出功能、上拉、速度等等。

③、STM32 有的 IO 可以作为其它外设引脚，也就是 IO 复用，如果要将 IO 作为其它外设引脚使用的话就需要设置 IO 的复用功能。

④、最后设置 GPIO 输出高电平或者低电平。

STM32 的 GPIO 初始化就是以上四步，那么会不会也适用于 I.MX6U 的呢？I.MX6U 的 GPIO 是不是也需要开启相应的时钟？是不是也可以设置复用功能？是不是也可以设置输出或输入、上下拉、速度等等这些？我们现在都不知道，只有去看 I.MX6U 的数据手册和参考手册才能知道，I.MX6U 的数据手册和参考手册我们已经放到了开发板光盘中了，I.MX6U 有 I.MX6UL 和 I.MX6ULL 两种，这两种型号基本是一样的，我们以 I.MX6UL 为例来讲解。I.MX6UL 的参考手册路径为：开发板光盘->1、I.MX6UL 芯片资料->IMX6UL 参考手册.pdf，I.MX6UL 的数据手册有三种，分别对应：车规级、工业级和商用级。从我们写代码的角度看，这三份数据手册一模一样的，做硬件的在选型的时候才需要注意一下，我们就用商用级的手册，商用级数据手册路径为：开发板光盘->1、I.MX6UL 芯片资料->IMX6UL 数据手册(商用级)pdf。带着上面四个疑问打开这两份手册，然后就是“啃”手册。

8.1.2 I.MX6U IO 命名

STM32 中的 IO 都是 PA0~15、PB0~15 这样命名的，I.MX6U 的 IO 是怎么命名的呢？打开 I.MX6UL 参考手册的第 30 章“Chapter 30: IOMUX Controller(IOMUXC)”，第 30 章的书签如图

8.1.2.1 所示：

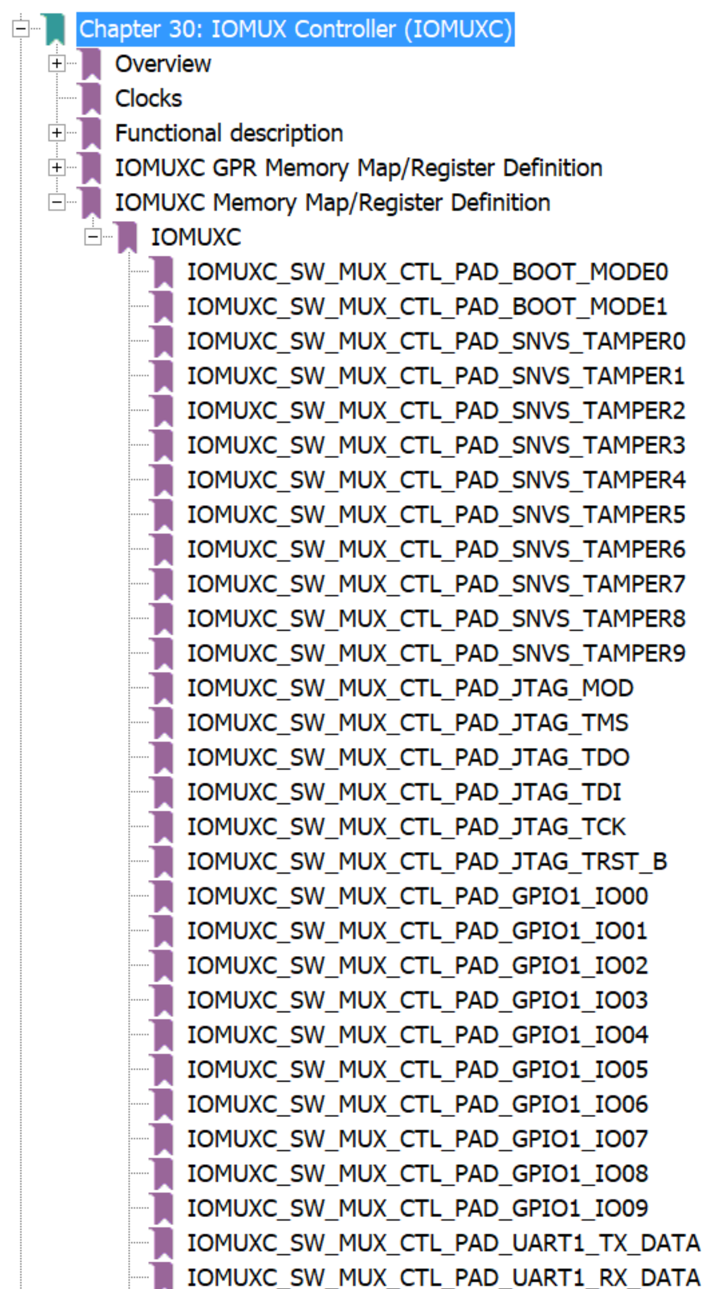
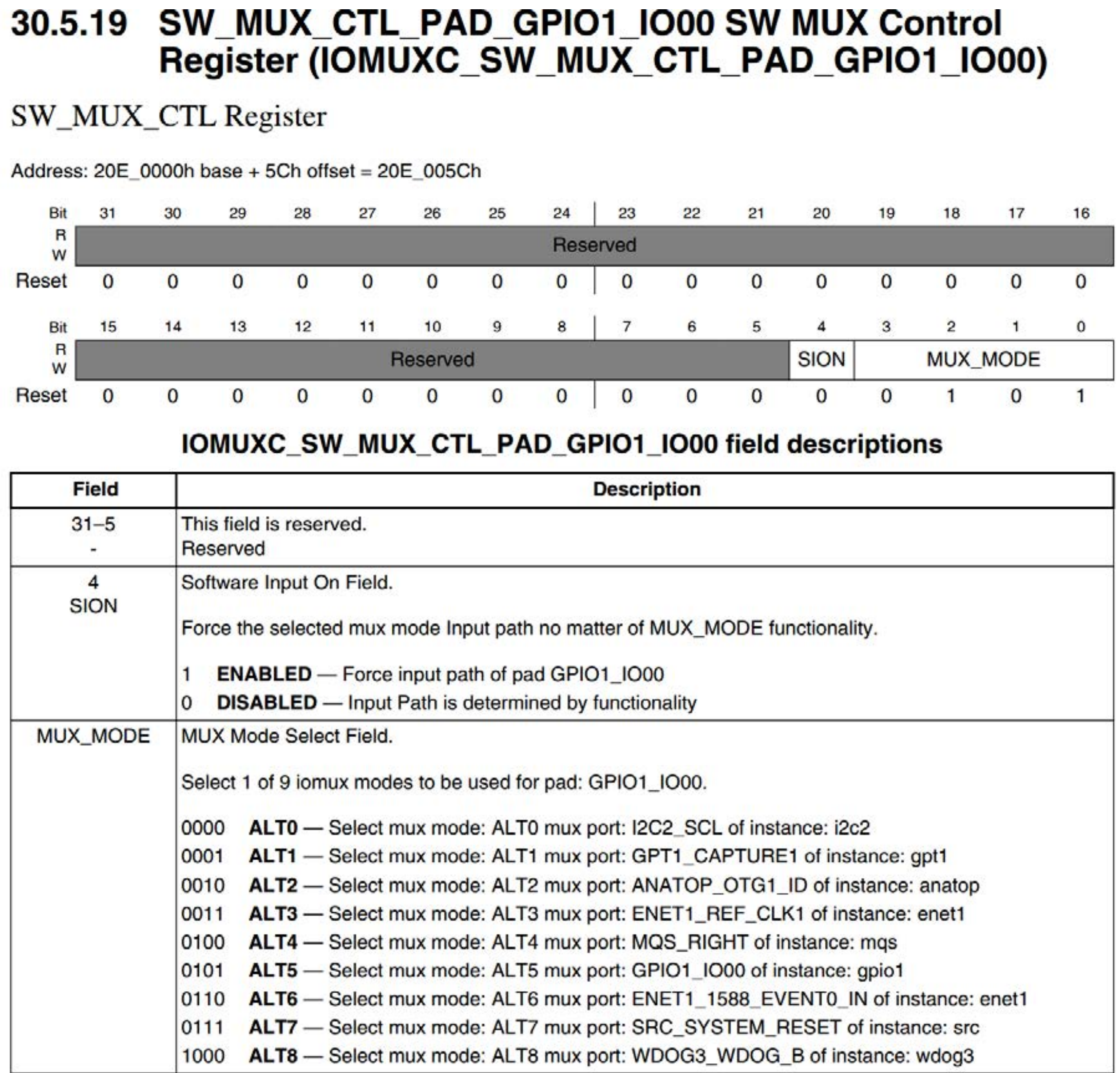


图 8.1.2.1 I.MX6U GPIO 命名

图 8.1.2.1 中的形如“IOMUXC_SW_MUC_CTL_PAD_GPIO1_IO00”的就是 GPIO 命名，命名形式就是“IOMUXC_SW_MUC_CTL_PAD_XX_XX”，后面的“XX_XX”就是 GPIO 命名，比如：GPIO1_IO01、UART1_TX_DATA、JTAG_MOD、SNVS_TAMPER1 等等。I.MX6U 的 GPIO 并不像 STM32 一样以 PA0~15 这样命名，他是根据某个 IO 所拥有的功能来命名的。比如我们一看到 GPIO1_IO01 就知道这个肯定能做 GPIO，看到 UART1_TX_DATA 肯定就知道这个 IO 肯定能做为 UART1 的发送引脚。“Chapter 30: IOMUX Controller(IOMUXC)”这一章列出了 I.MX6U 的所有 IO，如果你找遍第 30 章的书签，你会发现貌似 GPIO 只有 GPIO1_IO00~GPIO1_IO09，难道 I.MX6U 的 GPIO 只有这 10 个？显然不是的，我们知道 STM32 的很多 IO 是可以复用为其它功能的，那么 I.MX6U 的其它 IO 也是可以复用为 GPIO 功能。同样的，GPIO1_IO00~GPIO_IO09 也是可以复用为其它外设引脚的，接下来就是 I.MX6U IO 复用。

8.1.3 I.MX6U IO 复用

以 IO“IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00”为例，打开参考手册的 1329 页，如图 8.1.3.1 所示：



30.5.19 SW_MUX_CTL_PAD_GPIO1_IO00 SW MUX Control Register (IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00)

SW_MUX_CTL Register

Address: 20E_0000h base + 5Ch offset = 20E_005Ch

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00 field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad GPIO1_IO00 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	MUX Mode Select Field. Select 1 of 9 iomux modes to be used for pad: GPIO1_IO00. 0000 **ALT0** — Select mux mode: ALT0 mux port: I2C2_SCL of instance: i2c2 0001 **ALT1** — Select mux mode: ALT1 mux port: GPT1_CAPTURE1 of instance: gpt1 0010 **ALT2** — Select mux mode: ALT2 mux port: ANATOP_OTG1_ID of instance: anatop 0011 **ALT3** — Select mux mode: ALT3 mux port: ENET1_REF_CLK1 of instance: enet1 0100 **ALT4** — Select mux mode: ALT4 mux port: MQS_RIGHT of instance: mqs 0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO1_IO00 of instance: gpio1 0110 **ALT6** — Select mux mode: ALT6 mux port: ENET1_1588_EVENT0_IN of instance: enet1 0111 **ALT7** — Select mux mode: ALT7 mux port: SRC_SYSTEM_RESET of instance: src 1000 **ALT8** — Select mux mode: ALT8 mux port: WDOG3_WDOG_B of instance: wdog3

图 8.1.3.1 GPIO1_IO00 复用

从图 8.1.3.1 可以看到有个名为：IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00 的寄存器，寄存器地址为 0X020E005C，这个寄存器是 32 位的，但是只用到了最低 5 位，其中 bit0~bit3(MUX_MODE)就是设置 GPIO1_IO00 的复用功能的。GPIO1_IO00 一共可以复用为 9 种功能 IO，分别对应 ALT0~ALT8，其中 ALT5 就是作为 GPIO1_IO00。GPIO1_IO00 还可以作为 I2C2_SCL、GPT1_CAPTURE1、ANATOP_OTG1_ID 等。这个就是 I.MX6U 的 IO 复用，我们学习 STM32 的时候 STM32 的 GPIO 也是可以复用的。

再来看一个“IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA”这个 IO，这个 IO 对应的复用如图 8.1.3.2 所示：

32.6.17 SW_MUX_CTL_PAD_UART1_TX_DATA SW MUX Control Register (IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA)

SW_MUX_CTL Register

Address: 20E_0000h base + 84h offset = 20E_0084h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved											SION	MUX_MODE			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

IOMUXC_SW_MUX_CTL_PAD_UART1_TX_DATA field descriptions

Field	Description
31–5 -	This field is reserved. Reserved
4 SION	Software Input On Field. Force the selected mux mode Input path no matter of MUX_MODE functionality. 1 **ENABLED** — Force input path of pad UART1_TX_DATA 0 **DISABLED** — Input Path is determined by functionality
MUX_MODE	MUX Mode Select Field. Select 1 of 10 iomux modes to be used for pad: UART1_TX_DATA. 0000 **ALT0** — Select mux mode: ALT0 mux port: UART1_TX of instance: uart1 0001 **ALT1** — Select mux mode: ALT1 mux port: ENET1_RDATA02 of instance: enet1 0010 **ALT2** — Select mux mode: ALT2 mux port: I2C3_SCL of instance: i2c3 0011 **ALT3** — Select mux mode: ALT3 mux port: CSI_DATA02 of instance: csi 0100 **ALT4** — Select mux mode: ALT4 mux port: GPT1_COMPARE1 of instance: gpt1 0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO1_IO16 of instance: gpio1 1000 **ALT8** — Select mux mode: ALT8 mux port: SPDIF_OUT of instance: spdif 1001 **ALT9** — Select mux mode: ALT9 mux port: UART5_TX of instance: uart5

图 8.1.3.2 UART1_TX_DATA IO 复用

同样的，从图 8.1.3.2 可以看出，UART1_TX_DATA 可以复用为 8 种不同功能的 IO，分为 ALT0~ALT5 和 ALT8、ATL9，其中 ALT5 表示 UART1_TX_DATA 可以复用为 GPIO1_IO16。

由此可见，I.MX6U 的 GPIO 不止 GPIO1_IO00~GPIO1_IO09 这 10 个，其它的 IO 都可以复用为 GPIO 来使用。I.MX6U 的 GPIO 一共有 5 组：GPIO1、GPIO2、GPIO3、GPIO4 和 GPIO5，其中 GPIO1 有 32 个 IO，GPIO2 有 22 个 IO，GPIO3 有 29 个 IO、GPIO4 有 29 个 IO，GPIO5 最少，只有 12 个 IO，这样一共有 124 个 GPIO。如果只想看每个 IO 能复用什么外设的话可以直接查阅《IMX6UL 参考手册》的第 4 章“Chapter 4 External Signals and Pin Multiplexing”。如果我们要编写代码，设置某个 IO 的复用功能的话就需要查阅第 30 章“Chapter 30: IOMUX Controller(IOMUXC)”,第 30 章详细的列出了所有 IO 对应的复用配置寄存器。

至此我们就解决了 8.1.1 中的第 3 个疑问，那就是 I.MX6U 的 IO 是有复用功能的，和 STM32 一样，如果某个 IO 要作为某个外设引脚使用的话，是需要配置复用寄存器的。

8.1.4 I.MX6U IO 配置

细心的读者应该会发现在《I.MX6UL 参考手册》第 30 章“Chapter 30: IOMUX

Controller(IOMUXC)”的书签中，每一个 IO 会出现两次，它们的名字差别很小，不仔细看就看不出来，比如 GPIO1_IO00 有如下两个书签：

```
IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO00
IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00
```

上面两个都是跟 GPIO_IO00 有关的寄存器，名字上的区别就是红色部分，一个是“MUX”，一个是“PAD”。IOMUX_SW_MUX_CTL_PAD_GPIO1_IO00 我们前面已经说了，是用来配置 GPIO1_IO00 复用功能的，那么 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 是做什么的呢？找到这个书签对应的 1582 页，如图 8.1.4.1 所示：

30.5.182 SW_PAD_CTL_PAD_GPIO1_IO00 SW PAD Control Register (IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00)

SW_PAD_CTL Register

Address: 20E_0000h base + 2E8h offset = 20E_02E8h

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	Reserved															HYS
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	PUS		PUE	PKE	ODE	Reserved			SPEED		DSE			Reserved		SRE
Reset	0	0	0	1	0	0	0	0	1	0	1	1	0	0	0	0

IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 field descriptions

Field	Description
31–17 -	This field is reserved. Reserved
16 HYS	Hyst. Enable Field Select one out of next values for pad: GPIO1_IO00 0 **HYS_0_Hysteresis_Disabled** — Hysteresis Disabled 1 **HYS_1_Hysteresis_Enabled** — Hysteresis Enabled
15–14 PUS	Pull Up / Down Config. Field Select one out of next values for pad: GPIO1_IO00 00 **PUS_0_100K_Ohm_Pull_Down** — 100K Ohm Pull Down 01 **PUS_1_47K_Ohm_Pull_Up** — 47K Ohm Pull Up 10 **PUS_2_100K_Ohm_Pull_Up** — 100K Ohm Pull Up 11 **PUS_3_22K_Ohm_Pull_Up** — 22K Ohm Pull Up

图 8.1.4.1 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 寄存器

从图 8.1.4.1 中可以看出，IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 也是个寄存器，寄存器地址为 0X020E02E8。这也是个 32 位寄存器，但是只用到了其中的低 17 位，在看这写位的具体含义之前，先来看一下图 8.1.4.2 所示的 GPIO 功能图：

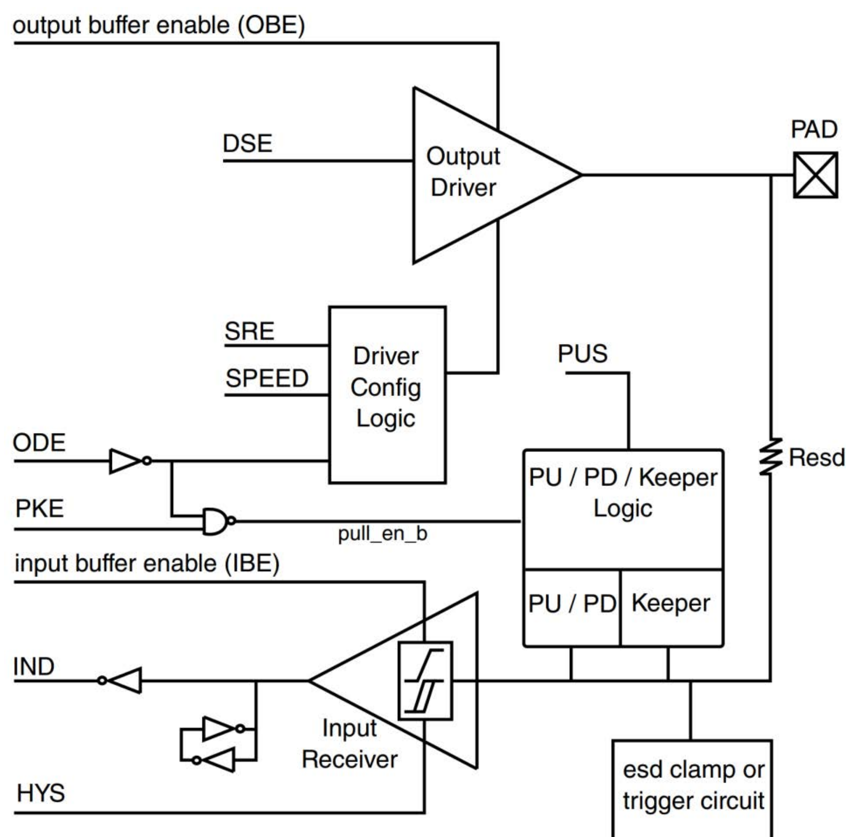


图 8.1.4.2 GPIO 功能图。

我们对照着图 8.1.4.2 来详细看一下寄存器 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 的各个位的含义：

HYS(bit16)：对应图 8.1.4.2 中 HYS，用来使能迟滞比较器，当 IO 作为输入功能的时候有效，用于设置输入接收器的施密特触发器是否使能。如果需要对输入波形进行整形的话可以使能此位。此位为 0 的时候禁止迟滞比较器，为 1 的时候使能迟滞比较器。

PUS(bit15:14)：对应图 8.1.4.2 中的 PUS，用来设置上下拉电阻的，一共有四种选项可以选择，如表 8.1.4.1 所示：

位设置	含义
00	100K 下拉
01	47K 上拉
10	100K 上拉
11	22K 上拉

表 8.1.4.1 上下拉设置

PUE(bit13)：图 8.1.4.2 没有给出来，当 IO 作为输入的时候，这个位用来设置 IO 使用上下拉还是状态保持器。当为 0 的时候使用状态保持器，当为 1 的时候使用上下拉。状态保持器在 IO 作为输入的时候才有用，顾名思义，就是当外部电路断电以后此 IO 口可以保持住以前的状态。

PKE(bit12)：对应图 8.1.4.2 中的 PKE，此为用来使能或者禁止上下拉/状态保持器功能，为 0 时禁止上下拉/状态保持器，为 1 时使能上下拉和状态保持器。

ODE(bit11)：对应图 8.1.4.2 中的 ODE，当 IO 作为输出的时候，此位用来禁止或者使能开路输出，此位为 0 的时候禁止开路输出，当此位为 1 的时候就使能开路输出功能。

SPEED(bit7:6)：对应图 8.1.4.2 中的 SPEED，当 IO 用作输出的时候，此位用来设置 IO 速度，设置如表 8.1.4.2 所示：

位设置	速度
00	低速 50M
01	中速 100M
10	中速 100M
11	最大速度 200M

表 8.1.4.2 速度配置

DSE(bit5:3)：对应图 8.1.4.2 中的 DSE，当 IO 用作输出的时候用来设置 IO 的驱动能力，总共有 8 个可选选项，如表 8.1.4.3 所示：

位设置	速度
000	输出驱动关闭
001	R0(3.3V 下 R0 是 260Ω，1.8V 下 R0 是 150Ω，接 DDR 的时候是 240Ω)
010	R0/2
011	R0/3
100	R0/4
101	R0/5
110	R0/6
111	R0/7

表 8.1.4.3 驱动能力设置

SRE(bit0)：对应图 8.1.4.2 中的 SRE，设置压摆率，当此位为 0 的时候是低压摆率，当为 1 的时候是高压摆率。这里的压摆率就是 IO 电平跳变所需要的时间，比如从 0 到 1 需要多少时间，时间越小波形就越陡，说明压摆率越高；反之，时间越多波形就越缓，压摆率就越低。如果你的产品要过 EMC 的话那就可以使用小的压摆率，因为波形缓和，如果你当前所使用的 IO 做高速通信的话就可以使用高压摆率。

通过上面的介绍，可以看出寄存器 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO00 是用来配置 GPIO1_IO00 的，包括速度设置、驱动能力设置、压摆率设置等等。至此我们就解决了 8.1.1 中的第 2 个疑问，那就是 I.MX6U 的 IO 是可以设置速度的、而且比 STM32 的设置要更多。但是我们没有看到如何设置 IO 为输入还是输出？IO 的默认电平如何设置等等，所以我们接着继续看。

8.1.5 I.MX6U GPIO 配置

IOMUXC_SW_MUX_CTL_PAD_XX_XX 和 IOMUXC_SW_PAD_CTL_PAD_XX_XX 这两种寄存器都是配置 IO 的，注意是 IO！不是 GPIO，GPIO 是一个 IO 众多复用功能中的一种。比如 GPIO1_IO00 这个 IO 可以复用为：I2C2_SCL、GPT1_CAPTURE1、ANATOP_OTG1_ID、ENET1_REF_CLK 、 MQS_RIGHT 、 GPIO1_IO00 、 ENET1_1588_EVENT0_IN 、SRC_SYSTEM_RESET 和 WDOG3_WDOG_B 这 9 个功能，GPIO1_IO00 是其中的一种，我们想要把 GPIO1_IO00 用作哪个外设就复用为哪个外设功能即可。如果我们要用 GPIO1_IO00 来点个灯、作为按键输入啥的就是使用其 GPIO(通用输入输出)的功能。将其复用为 GPIO 以后还需要对其 GPIO 的功能进行配置，关于 I.MX6U 的 GPIO 请参考《IMX6UL 参考手册》的第 26 章“Chapter 26 General Purpose Input/Ouput(GPIO)”，GPIO 结构如图 8.1.5.1 所示：

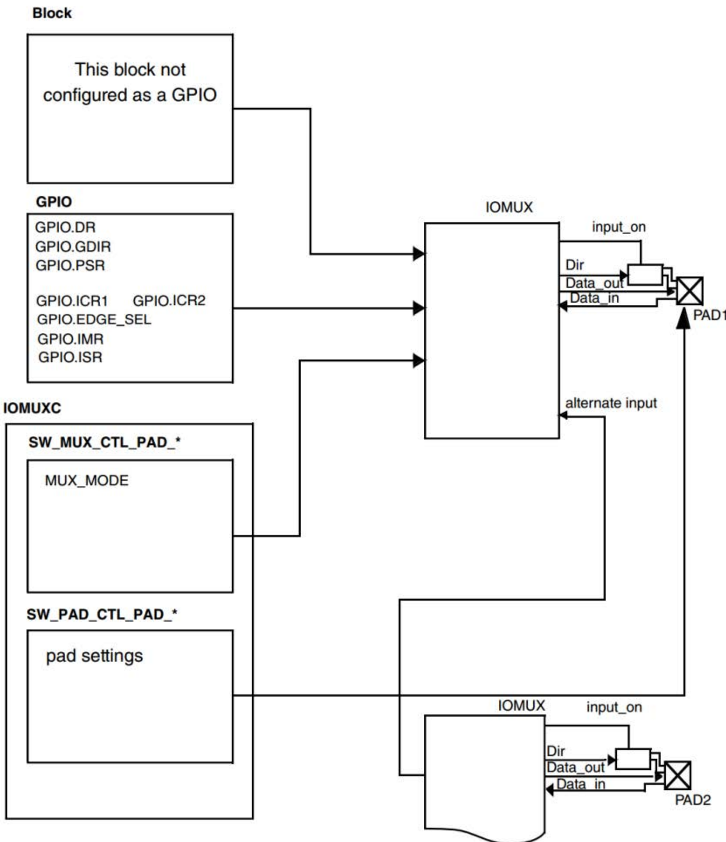
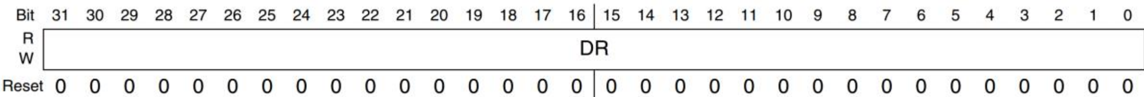


图 8.1.5.1 GPIO 结构图

在图 8.1.5.1 的左下角的 IOMUXC 框图里面就有 SW_MUX_CTL_PAD_*和 SW_PAD_CTL_PAD_*两种寄存器。这两种寄存器前面说了用来设置 IO 的复用功能和 IO 属性配置。左上角部分的 GPIO 框图就是，当 IO 用作 GPIO 的时候需要设置的寄存器，一共有八个：DR、GDIR、PSR、ICR1、ICR2、EDGE_SEL、IMR 和 ISR。前面我们说了 I.MX6U 一共有 GPIO1~GPIO5 共五组 GPIO，每组 GPIO 都有这 8 个寄存器。我们来看一下这 8 个寄存器都是什么含义。

首先来看一下 DR 寄存器，此寄存器是数据寄存器，结构图如图 8.1.5.2 所示：



GPIOx_DR field descriptions

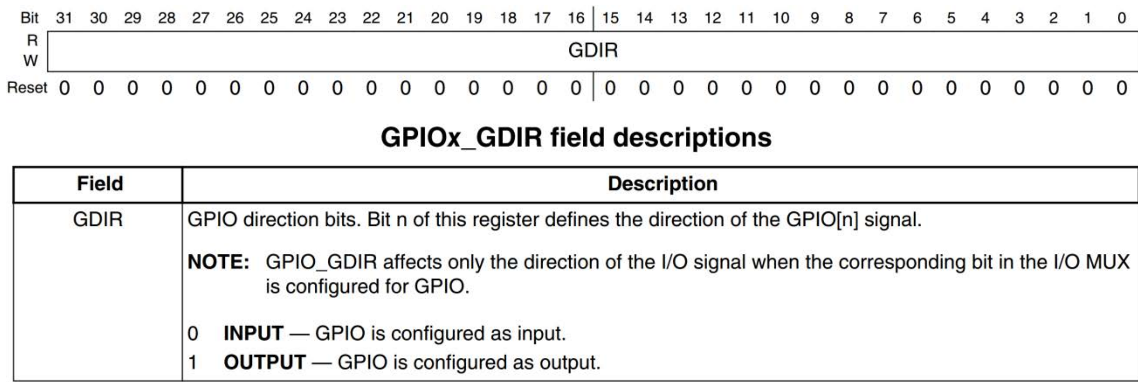
Field	Description
DR	Data bits. This register defines the value of the GPIO output when the signal is configured as an output (GDIR[n]=1). Writes to this register are stored in a register. Reading GPIO_DR returns the value stored in the register if the signal is configured as an output (GDIR[n]=1), or the input signal's value if configured as an input (GDIR[n]=0). **NOTE:** The I/O multiplexer must be configured to GPIO mode for the GPIO_DR value to connect with the signal. Reading the data register with the input path disabled always returns a zero value.

图 8.1.5.2 DR 寄存器结构图

此寄存器是 32 位的，一个 GPIO 组最大只有 32 个 IO，因此 DR 寄存器中的每个位都对应一个 GPIO。当 GPIO 被配置为输出功能以后，向指定的位写入数据那么相应的 IO 就会输出相应的高低电平，比如要设置 GPIO1_IO00 输出高电平，那么就应该设置 GPIO1.DR=1。当 GPIO

被配置为输入模式以后，此寄存器就保存着对应 IO 的电平值，每个位对对应一个 GPIO，例如，当 GPIO1_IO00 这个引脚接地的话，那么 GPIO1.DR 的 bit0 就是 0。

看完 DR 寄存器，接着看 GDIR 寄存器，这是方向寄存器，用来设置某个 GPIO 的工作方向的，即输入/输出，GDIR 寄存器结构如图 8.1.5.3 所示：



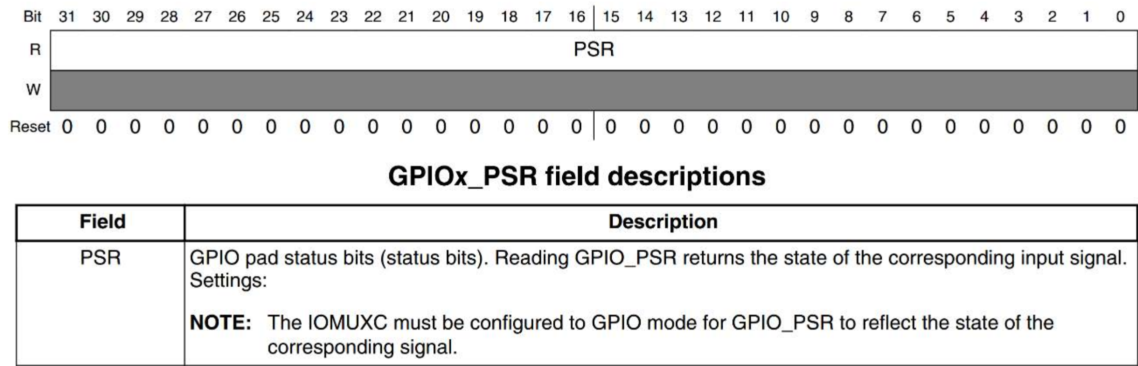
GPIOx_GDIR field descriptions

Field	Description
GDIR	GPIO direction bits. Bit n of this register defines the direction of the GPIO[n] signal. **NOTE:** GPIO_GDIR affects only the direction of the I/O signal when the corresponding bit in the I/O MUX is configured for GPIO. 0 **INPUT** — GPIO is configured as input. 1 **OUTPUT** — GPIO is configured as output.

图 8.1.5.3 GDIR 寄存器

GDIR 寄存器也是 32 位的，此寄存器用来设置某个 IO 的工作方向，是输入还是输出。同样的，每个 IO 对应一个位，如果要设置 GPIO 为输入的话就设置相应的位为 0，如果要设置为输出的话就设置为 1。比如要设置 GPIO1_IO00 为输入，那么 GPIO1.GDIR=0;

接下来看 PSR 寄存器，这是 GPIO 状态寄存器，如图 8.1.5.4 所示：



GPIOx_PSR field descriptions

Field	Description
PSR	GPIO pad status bits (status bits). Reading GPIO_PSR returns the state of the corresponding input signal. Settings: **NOTE:** The IOMUXC must be configured to GPIO mode for GPIO_PSR to reflect the state of the corresponding signal.

图 8.1.5.4 PSR 状态寄存器

同样的 PSR 寄存器也是一个 GPIO 对应一个位，读取相应的位即可获取对应的 GPIO 的状态，也就是 GPIO 的高低电平值。功能和输入状态下的 DR 寄存器一样。

接下来看 ICR1 和 ICR2 这两个寄存器，都是中断控制寄存器，ICR1 用于配置低 16 个 GPIO，ICR2 用于配置高 16 个 GPIO，ICR1 寄存器如图 8.1.5.5 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	ICR15		ICR14		ICR13		ICR12		ICR11		ICR10		ICR9		ICR8	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	ICR7		ICR6		ICR5		ICR4		ICR3		ICR2		ICR1		ICR0	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_ICR1 field descriptions

Field	Description
31–30 ICR15	Interrupt configuration 1 fields. This register controls the active condition of the interrupt function for GPIO interrupt 15. Settings: Bits ICRn[1:0] determine the interrupt condition for signal n as follows: 00 **LOW_LEVEL** — Interrupt n is low-level sensitive. 01 **HIGH_LEVEL** — Interrupt n is high-level sensitive. 10 **RISING_EDGE** — Interrupt n is rising-edge sensitive. 11 **FALLING_EDGE** — Interrupt n is falling-edge sensitive.

图 8.1.5.5 ICR1 寄存器

ICR1 用于 IO0~15 的配置， ICR2 用于 IO16~31 的配置。ICR1 寄存器中一个 GPIO 用两个位，这两个位用来配置中断的触发方式，和 STM32 的中断很类似，可配置的选线如表 8.1.5.1 所示：

位设置	速度
00	低电平触发
01	高电平触发
10	上升沿触发
11	下降沿触发

表 8.1.5.1 中断触发配置

以 GPIO1_IO15 为例，如果要设置 GPIO1_IO15 为上升沿触发中断，那么 GPIO1.ICR1=2<<30，如果要设置 GPIO1 的 IO16~31 的话就需要设置 ICR2 寄存器了。

接下来看 IMR 寄存器，这是中断屏蔽寄存器，如图 8.1.5.6 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	IMR																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

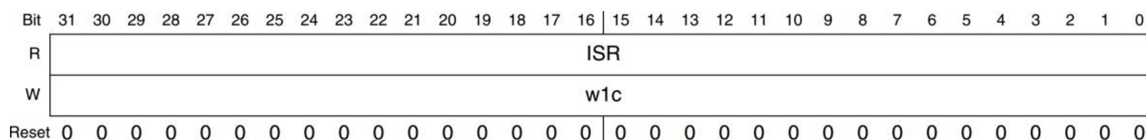
GPIOx_IMR field descriptions

Field	Description
IMR	Interrupt Mask bits. This register is used to enable or disable the interrupt function on each of the 32 GPIO signals. Settings: Bit IMR[n] (n=0...31) controls interrupt n as follows: 0 **UNMASKED** — Interrupt n is disabled. 1 **MASKED** — Interrupt n is enabled.

图 8.1.5.6 IMR 寄存器

IMR 寄存器也是一个 GPIO 对应一个位，IMR 寄存器用来控制 GPIO 的中断禁止和使能，如果使能某个 GPIO 的中断，那么设置相应的位为 1 即可，反之，如果要禁止中断，那么就设置相应的位为 0 即可。例如，要使能 GPIO1_IO00 的中断，那么就可以设置 GPIO1.MIR=1 即可。

接下来看寄存器 ISR，ISR 是中断状态寄存器，寄存器如图 8.1.5.7 所示：



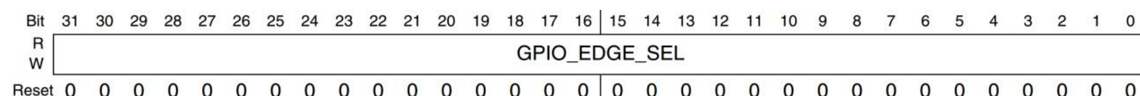
GPIOx_ISR field descriptions

Field	Description
ISR	Interrupt status bits - Bit n of this register is asserted (active high) when the active condition (as determined by the corresponding ICR bit) is detected on the GPIO input and is waiting for service. The value of this register is independent of the value in GPIO_IMR. When the active condition has been detected, the corresponding bit remains set until cleared by software. Status flags are cleared by writing a 1 to the corresponding bit position.

图 8.1.5.7 ISR 寄存器

ISR 寄存器也是 32 位寄存器，一个 GPIO 对应一个位，只要某个 GPIO 的中断发生，那么 ISR 中相应的位就会被置 1。所以，我们可以通过读取 ISR 寄存器来判断 GPIO 中断是否发生，相当于 ISR 中的这些位就是中断标志位。当我们处理完中断以后，必须清除中断标志位，清除方法就是向 ISR 中相应的位写 1，也就是写 1 清零。

最后来看一下 EDGE_SEL 寄存器，这是边沿选择寄存器，寄存器如图 8.1.5.8 所示：



GPIOx_EDGE_SEL field descriptions

Field	Description
GPIO_EDGE_SEL	Edge select. When GPIO_EDGE_SEL[*n*] is set, the GPIO disregards the ICR[*n*] setting, and detects any edge on the corresponding input signal.

图 8.1.5.8 EDGE_SEL 寄存器

EDGE_SEL 寄存器用来设置边沿中断，这个寄存器会覆盖 ICR1 和 ICR2 的设置，同样是一个 GPIO 对应一个位。如果相应的位被置 1，那么就相当与设置了对应的 GPIO 是上升沿和下降沿(双边沿)触发。例如，我们设置 GPIO1.EDGE_SEL=1，那么就表示 GPIO1_IO01 是双边沿触发中断，无论 GFPIO1_CR1 的设置为多少，都是双边沿触发。

关于 GPIO 的寄存器就讲解到这里，因为 GPIO 是最常用的功能，我们详细的讲解了 GPIO 的 8 个寄存器。至此我们就解决了 8.1.1 中的第 3 个和第 4 个疑问，那就是 I.MX6U 的 IO 是需要配置和输出的、是可以设置输出高低电平，也可以读取 GPIO 对应的电平。

8.1.6 I.MX6U GPIO 时钟使能

还有最后一个疑问，那就是 I.MX6U 的 GPIO 是否需要使能时钟？STM32 的每个外设都有一个外设时钟，GPIO 也不例外，要使用某个外设，必须要先使能对应的时钟。I.MX6U 其实也一样的，每个外设的时钟都可以独立的使能或禁止，这样可以关闭掉不使用的外设时钟，起到省电的目的。如果要使用某个外设的话必须要先使能其时钟。I.MX6U 的系统时钟参考《I.MX6UL 参考手册》的第 18 章“Chapter 18: Clock Controller Module(CCM)”，这一个章主要讲解 I.MX6U 的时钟系统，很复杂。我们先不研究 I.MX6U 的时钟系统，我们只看一下 CCM 里面的外设时钟使能寄存器。CMM 有 CCM_CCGR0~CCM_CCGR6 这 7 个寄存器，这 7 个寄存器控制着 I.MX6U 的所有外设时钟开关，我们以 CCM_CCGR0 为例来看一下如何禁止或使能一个外设的时钟，CCM_CCGR0 结构体如图 8.1.6.1 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	CG15		CG14		CG13		CG12		CG11		CG10		CG9		CG8	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	CG7		CG6		CG5		CG4		CG3		CG2		CG1		CG0	
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

CCM_CCGR0 field descriptions

Field	Description
31–30 CG15	gpio2_clocks (gpio2_clk_enable)
29–28 CG14	uart2 clock (uart2_clk_enable)
27–26 CG13	gpt2 serial clocks (gpt2_serial_clk_enable)
25–24 CG12	dcic1 clocks (dcic1_clk_enable)gpt2 bus clocks (gpt2_bus_clk_enable)
23–22 CG11	CPU debug clocks (arm_dbg_clk_enable)
21–20 CG10	can2_serial clock (can2_serial_clk_enable)
19–18 CG9	can2 clock (can2_clk_enable)
17–16 CG8	can1_serial clock (can1_serial_clk_enable)
15–14 CG7	can1 clock (can1_clk_enable)
13–12 CG6	caam_wrapper_ipg clock (caam_wrapper_ipg_enable)
11–10 CG5	caam_wrapper_aclk clock (caam_wrapper_aclk_enable)
9–8 CG4	caam_secure_mem clock (caam_secure_mem_clk_enable)
7–6 CG3	asrc clock (asrc_clk_enable)
5–4 CG2	apbhdma hclk clock (apbhdma_hclk_enable)
3–2 CG1	aips_tz2 clocks (aips_tz2_clk_enable)
CG0	aips_tz1 clocks (aips_tz1_clk_enable)

图 8.1.6.1 CCM_CCGR0 寄存器

CCM_CCGR0 是个 32 为寄存器，其中每 2 位控制一个外设的时钟，比如 bit31:30 控制着 GPIO2 的外设时钟，两个位就有 4 中操作方式，如表 8.1.6.1 所示：

位设置	时钟控制
00	所有模式下都关闭外设时钟。
01	只有在运行模式下打开外设时钟，等待模式和停止模式下均关闭外设时钟。
10	未使用(保留)。
11	除了停止模式以外，其他所有模式下时钟都打开。

表 8.1.6.1 外设时钟控制

根据表 8.1.6.1 中的位设置，如果我们要打开 GPIO2 的外设时钟，那么只需要设置 CCM_CCGR0 的 bit31 和 bit30 都为 1 即可，也就是 CCM_CCGR0=3 << 30。反之，如果要关闭 GPIO2 的外设时钟，那就设置 CCM_CCGR0 的 bit31 和 bit30 都为 0 即可。

CCM_CCGR0~CCM_CCGR6 这 7 个寄存器操作都是类似的，只是不同的寄存器对应不同的外设时钟而已，为了方便开发，本教程后面所有的例程将 I.MX6U 的所有外设时钟都打开了。至此我们就解决了 8.1.1 中的所有问题都解决了，I.MX6U 的每个外设的时钟都可以独立的禁止和使能，这个和 STM32 是一样。 总结一下，要将 I.MX6U 的 IO 作为 GPIO 使用，我们需要一下几步：

①、使能 GPIO 对应的时钟。

②、设置寄存器 IOMUXC_SW_MUX_CTL_PAD_XX_XX，设置 IO 的复用功能，使其复用为 GPIO 功能。

③、设置寄存器 IOMUXC_SW_PAD_CTL_PAD_XX_XX，设置 IO 的上下拉、速度等等。

④、第②步已经将 IO 复用为了 GPIO 功能，所以需要配置 GPIO，设置输入/输出、是否使用中断、默认输出电平等。

8.2 硬件原理分析

打开 I.MX6U-ALPHA 开发板底板原理图，底板原理图和核心板原理图都放到了开发板光盘中，路径为: 开发板光盘->2、开发板原理图->IMX6UL_ALPHA_V1.0(底板原理图)。I.MX6U-ALPHA 开发板上有一个 LED 灯，原理图如下 8.2.1 所示：

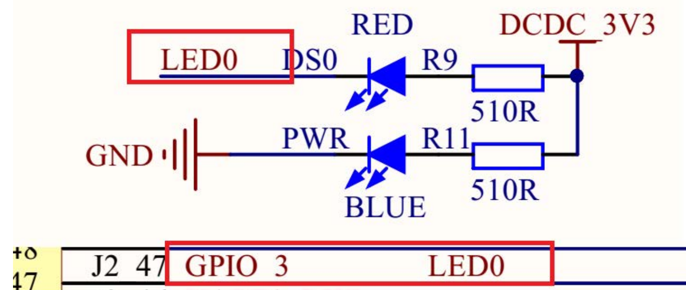


图 8.2.1 LED 原理图

从图 8.2.1 可以看出，LED0 接到了 GPIO_3 上，GPIO_3 就是 GPIO1_IO03，当 GPIO1_IO03 输出低电平(0)的时候发光二极管 LED0 就会导通点亮，当 GPIO1_IO03 输出高电平(1)的时候发光二极管 LED0 不会导通，因此 LED0 也就不会点亮。所以 LED0 的亮灭取决于 GPIO1_IO03 的输出电平，输出 0 就亮，输出 1 就灭。

8.3 实验程序编写

按照 8.1 小节中讲的，我们需要对 GPIO1_IO03 做如下设置：

1、使能 GPIO1 时钟

GPIO1 的时钟由 CCM_CCGR1 的 bit27 和 bit26 这两个位控制，将这两个位都设置位 11 即可。本教程所有例程已经将 I.MX6U 的所有外设时钟都已经打开了，因此这一步可以不用做。

2、设置 GPIO1_IO03 的复用功能

找到 GPIO1_IO03 的复用寄存器“IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03”的地址为 0X020E0068，然后设置此寄存器，将 GPIO1_IO03 这个 IO 复用为 GPIO 功能，也就是 ALT5。

3、配置 GPIO1_IO03

找到 GPIO1_IO03 的配置寄存器“IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03”的地址为 0X020E02F4，根据实际使用情况，配置此寄存器。

4、设置 GPIO

我们已经将 GPIO1_IO03 复用为了 GPIO 功能，所以我们需要配置 GPIO。找到 GPIO3 对应的 GPIO 组寄存器地址，在《IMX6UL 参考手册》的 1154 页，如图 8.3.1 所示：

20A_4000	GPIO data register (GPIO3_DR)	32	R/W	0000_0000h	26.5.1/1155
20A_4004	GPIO direction register (GPIO3_GDIR)	32	R/W	0000_0000h	26.5.2/1156
20A_4008	GPIO pad status register (GPIO3_PSR)	32	R	0000_0000h	26.5.3/1156
20A_400C	GPIO interrupt configuration register1 (GPIO3_ICR1)	32	R/W	0000_0000h	26.5.4/1157
20A_4010	GPIO interrupt configuration register2 (GPIO3_ICR2)	32	R/W	0000_0000h	26.5.5/1161
20A_4014	GPIO interrupt mask register (GPIO3_IMR)	32	R/W	0000_0000h	26.5.6/1164
20A_4018	GPIO interrupt status register (GPIO3_ISR)	32	w1c	0000_0000h	26.5.7/1165
20A_401C	GPIO edge select register (GPIO3_EDGE_SEL)	32	R/W	0000_0000h	26.5.8/1166

图 8.3.1 GPIO3 对应的 GPIO 寄存器地址

本实验中 GPIO1_IO03 是作为输出功能的，因此 GPIO3_GDIR 的 bit3 要设置为 1，表示输出。

5、控制 GPIO 的输出电平

经过前面几步，GPIO1_IO03 已经配置好了，只需要向 GPIO3_DR 寄存器的 bit3 写入 0 即可控制 GPIO1_IO03 输出低电平，打开 LED，向 bit3 写入 1 可控制 GPIO1_IO03 输出高电平，关闭 LED。

本实验完整工程在开发板光盘中，路径为：开发板光盘-> 1、例程源码-> 1、裸机例程-> 1_leds，如果要打开这个工程的话一定要将“1_leds”整个文件夹复制到一个没有中文路径的目录中，否则直接打开工程可能会报错。

所有的裸机实验我们都在 Ubuntu 下完成，使用 VSCode 编辑器！

所有的裸机实验我们都在 Ubuntu 下完成，使用 VSCode 编辑器！

所有的裸机实验我们都在 Ubuntu 下完成，使用 VSCode 编辑器！

既然是实验，肯定要自己动手创建工程，新建一个名为“1_leds”的文件夹，然后在“1_leds”这个目录下新建一个名为“led.s”的汇编文件和一个名为“.vscode”的目录，创建好以后“1_leds”文件夹如图 8.3.2 所示：

```
zuozhongkai@ubuntu:~/1_leds$ ls -a
.  ..  led.s  .vscode
zuozhongkai@ubuntu:~/1_leds$
```

图 8.3.2 新建的 1_leds 工程文件夹

图 8.3.2 中.vscode 文件夹里面存放 VSCode 的工程文件，led.s 就是我们新建的汇编文件，我们稍后会在 led.s 这个文件中编写汇编程序。使用 VSCode 打开 1_leds 这个文件夹，打开以后如图 8.3.3 所示：

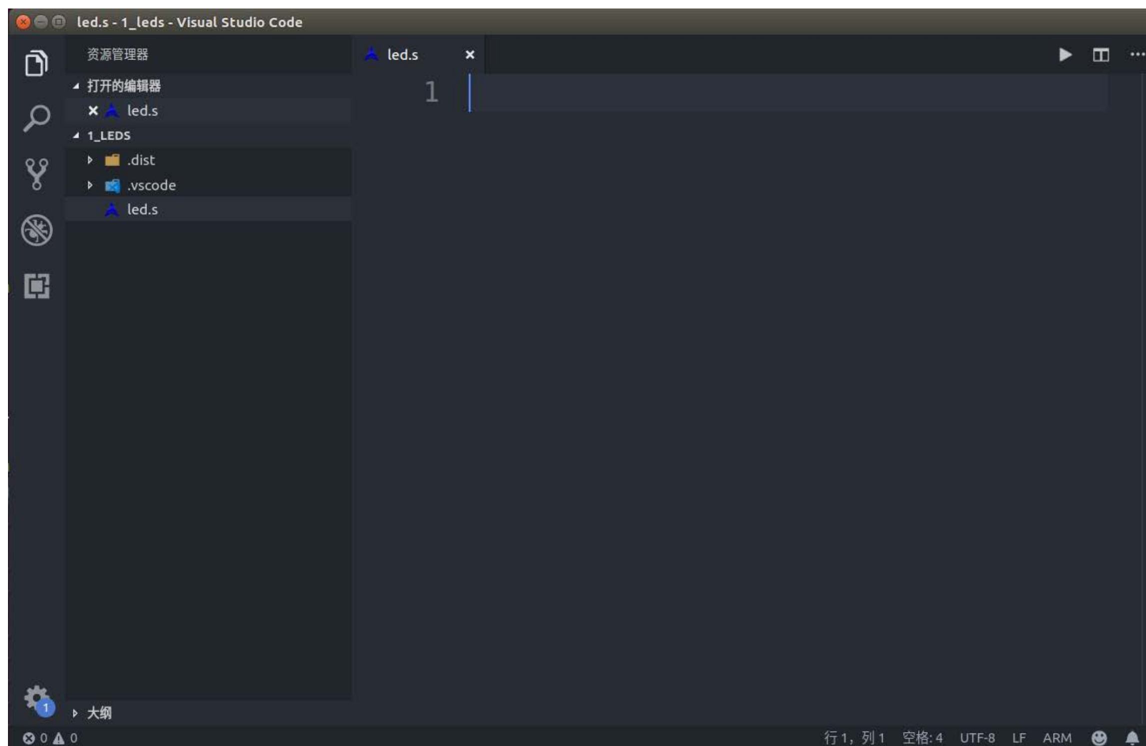


图 8.3.3 VSCode 工程

在 led.s 中输入如下代码：

示例代码 8.3.1 led.s 文件源码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : led.s
作者      : 左忠凯
版本      : V1.0
描述      : 裸机实验 1 汇编点灯
            使用汇编来点亮开发板上的 LED 灯，学习和掌握如何用汇编语言来
            完成对 I.MX6U 处理器的 GPIO 初始化和控制。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/3 左忠凯创建
***************************************************************/
1
2  .global _start  /* 全局标号 */
3
4  /*
5   * 描述：  _start 函数，程序从此函数开始执行此函数完成时钟使能、
6   *          GPIO 初始化、最终控制 GPIO 输出低电平来点亮 LED 灯。
7   */
8  _start:
9   /* 例程代码 */
10  /* 1、使能所有时钟 */
```

```
11 ldr r0, =0X020C4068     /* 寄存器 CCGR0 */
12 ldr r1, =0XFFFFFFFF
13 str r1, [r0]
14
15 ldr r0, =0X020C406C     /* 寄存器 CCGR1 */
16 str r1, [r0]
17
18 ldr r0, =0X020C4070     /* 寄存器 CCGR2 */
19 str r1, [r0]
20
21 ldr r0, =0X020C4074     /* 寄存器 CCGR3 */
22 str r1, [r0]
23
24 ldr r0, =0X020C4078     /* 寄存器 CCGR4 */
25 str r1, [r0]
26
27 ldr r0, =0X020C407C     /* 寄存器 CCGR5 */
28 str r1, [r0]
29
30 ldr r0, =0X020C4080     /* 寄存器 CCGR6 */
31 str r1, [r0]
32
33
34 /* 2、设置 GPIO1_IO03 复用为 GPIO1_IO03 */
35 ldr r0, =0X020E0068 /* 将寄存器 SW_MUX_GPIO1_IO03_BASE 加载到 r0 中 */
36 ldr r1, =0X5       /* 设置寄存器 SW_MUX_GPIO1_IO03_BASE 的 MUX_MODE 为 5 */
37 str r1,[r0]
38
39 /* 3、配置 GPIO1_IO03 的 IO 属性
40  *bit 16:0 HYS 关闭
41  *bit [15:14]: 00 默认下拉
42  *bit [13]: 0 kepper 功能
43  *bit [12]: 1 pull/keeper 使能
44  *bit [11]: 0 关闭开路输出
45  *bit [7:6]: 10 速度 100Mhz
46  *bit [5:3]: 110 R0/6 驱动能力
47  *bit [0]: 0 低转换率
48  */
49 ldr r0, =0X020E02F4  /*寄存器 SW_PAD_GPIO1_IO03_BASE */
50 ldr r1, =0X10B0
51 str r1,[r0]
52
53 /* 4、设置 GPIO1_IO03 为输出 */
```

```
54  ldr r0, =0X0209C004  /*寄存器 GPIO1_GDIR */
55  ldr r1, =0X0000008
56  str r1,[r0]
57
58  /* 5、打开 LED0
59   * 设置 GPIO1_IO03 输出低电平
60   */
61  ldr r0, =0X0209C000 /*寄存器 GPIO1_DR */
62  ldr r1, =0
63  str r1,[r0]
64
65 /*
66  * 描述：  loop 死循环
67  */
68 loop:
69      b loop
```

我们来详细的分析一下上面的汇编代码，我们以后分析代码都根据行号来分析。

第 2 行定义了一个全局标号_start，代码就是从_start 这个标号开始顺序往下执行的。

第 11 行使用 ldr 指令向寄存器 r0 写入 0X020C4068，也就是 r0=0X020C4068，这个是 CCM_CCGR0 寄存器的地址。

第 12 行使用 ldr 指令向寄存器 r1 写入 0XFFFFFFFF，也就是人 r1=0XFFFFFFFF。因为我们要开启所有的外设时钟，因此 CCM_CCGR0~CCM_CCGR6 所有的寄存器都 32 为位都要置 1，也就是写入 0XFFFFFFFF。

第 13 行使用 str 将 r1 中的值写入到 r0 所保存的地址中去，也就是给 0X020C4068 这个地址写入 0XFFFFFFFF，相当于 CCM_CCGR0=0XFFFFFFFF，就是打开 CCM_CCGR0 寄存器所控制的所有外设时钟。

第 15~31 行都是向 CCM_CCGRX(X=1~6)寄存器写入 0XFFFFFFFF。这样我就通过汇编代码使能了 I.MX6U 的所有外设时钟。

第 35~37 行是设置 GPIO1_IO03 的复用功能，GPIO1_IO03 的复用寄存器地址为 0X020E0068，寄存器 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 的 MUX_MODE 设置为 5 就是将 GPIO1_IO03 设置为 GPIO。

第 49~51 行是设置 GPIO1_IO03 的配置寄存器，也就是寄存器 IOMUX_SW_PAD_CTL_PAD_GPIO1_IO03 的值，此寄存器地址为 0X020E02F4，代码里面已经给出了这个寄存器详细的位设置。

第 54~63 行是设置 GPIO 功能，经过上面几步操作，GPIO1_IO03 这个 IO 已经被配置为了 GPIO 功能，所以还需要设置跟 GPIO 有关的寄存器。第 54~56 行是设置 GPIO1->GDIR 寄存器，将 GPIO1_IO03 设置为输出模式，也就是寄存器的 GPIO1_GDIR 的 bit3 置 1。

第 61~63 行设置 GPIO1->DR 寄存器，也就是设置 GPIO1_IO03 的输出，我们要点亮开发板上的 LED0，那么 GPIO1_IO03 就必须输出低电平，所以这里设置 GPIO1_DR 寄存器为 0。

第 68~69 行是死循环，通过 b 指令，CPU 重复不断的跳到 loop 函数执行，进入一个死循环。

8.4 编译下载验证

8.4.1 编译代码

如果你是在 Windows 下使用 Source Insight 编写的代码，就需要通过 FileZilla 将编写好的代码发送的 Ubuntu 中去编译，FileZilla 的使用参考 4.1 小节。因为我们现在是直接在 Ubuntu 下使用 VSCode 编译的代码，所以不需要使用 FileZilla 将代码发送到 Ubuntu 下，可以直接进行编译，在编译之前我们先了解几个编译工具。

1、arm-linux-gnueabihf-gcc 编译文件

我们是要编译出在 ARM 开发板上运行的可执行文件，所以要使用我们在 4.3 小节安装的交叉编译器 arm-linux-gnueabihf-gcc 来编译。因此本试验就一个 led.s 源文件，所以编译比较简单。先将 led.s 编译为对应的.o 文件，在终端中输入如下命令：

```
arm-linux-gnueabihf-gcc -g -c led.s -o led.o
```

上述命令就是将 led.s 编译为 led.o，其中“-g”选项是产生调试信息，GDB 能够使用这些调试信息进行代码调试。“-c”选项是编译源文件，但是不链接。“-o”选项是指定编译产生的文件名字，这里我们指定 led.s 编译完成以后的文件名字为 led.o。执行上述命令以后就会编译生成一个 led.o 文件，如图 8.4.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.o  led.s  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.1 编译生成 led.o 文件

图 8.4.1.1 中 led.o 文件并不是我们可以下载到开发板中运行的文件，一个工程中所有的 C 文件和汇编文件都会编译生成一个对应的.o 文件，我们需要将这.o 文件链接起来组合成可执行文件。

2、arm-linux-gnueabihf-ld 链接文件

arm-linux-gnueabihf-ld 用来将众多的.o 文件链接到一个指定的链接位置。我们在学习 SMT32 的时候基本就没有听过“链接”这个词，我们一般用 MDK 编写好代码，然后点击“编译”，MDK 或者 IAR 就会自动帮我们编译好整个工程，最后再点击“下载”就可以将代码下载到开发板中。这是因为链接这个操作 MDK 或者 IAR 已经帮你做好了，后面我就以 MDK 为例给大家讲解。大家可以打开一个 STM32 的工程，然后编译一下，肯定能找到很多.o 文件，如图 8.4.1.2 所示：

名称	修改日期	类型	大小
stm32f10x_dbgmcu.crf	2019-01-17 1:21	CRF 文件	341 KB
stm32f10x_dbgmcu.d	2019-01-17 1:21	D 文件	2 KB
stm32f10x_dbgmcu.o	2019-01-17 1:21	O 文件	376 KB
stm32f10x_gpio.crf	2019-01-17 1:21	CRF 文件	345 KB
stm32f10x_gpio.d	2019-01-17 1:21	D 文件	2 KB
stm32f10x_gpio.o	2019-01-17 1:21	O 文件	400 KB
stm32f10x_it.crf	2019-01-17 1:21	CRF 文件	341 KB
stm32f10x_it.d	2019-01-17 1:21	D 文件	2 KB
stm32f10x_it.o	2019-01-17 1:21	O 文件	384 KB
stm32f10x_rcc.crf	2019-01-17 1:21	CRF 文件	348 KB
stm32f10x_rcc.d	2019-01-17 1:21	D 文件	2 KB
stm32f10x_rcc.o	2019-01-17 1:21	O 文件	421 KB
stm32f10x_usart.crf	2019-01-17 1:21	CRF 文件	347 KB
stm32f10x_usart.d	2019-01-17 1:21	D 文件	2 KB
stm32f10x_usart.o	2019-01-17 1:21	O 文件	416 KB

图 8.4.1.2 STM32 编译生成的.o 文件

图 8.4.1.2 中的这些.o 文件肯定会被 MDK 链接到某个地址去，如果使用 MDK 开发 STM32 的话肯定对图 8.4.1.3 所示界面很熟悉：

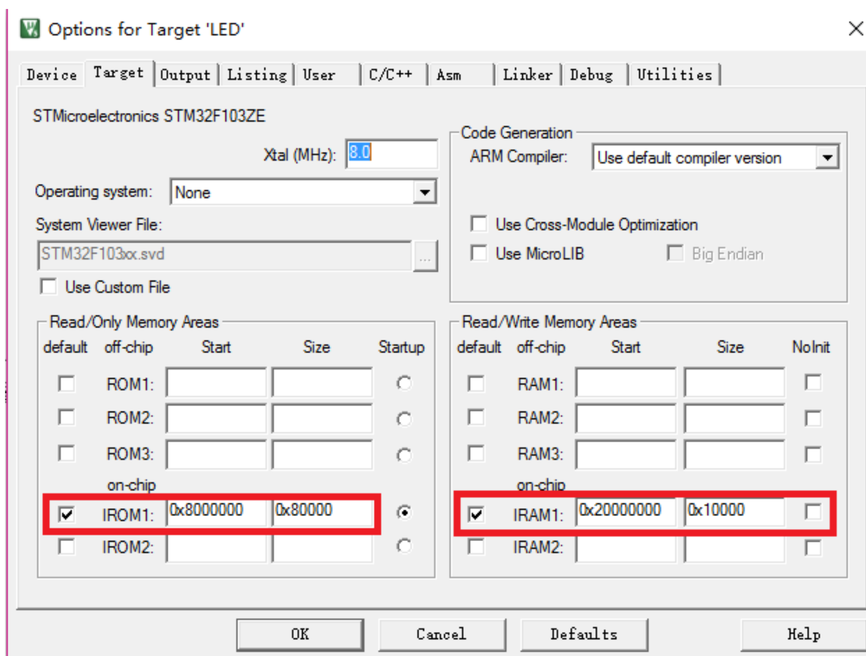


图 8.4.1.3 STM32 配置界面

图 8.4.1.3 中左侧的 IROM1 我们都知道是设置 STM32 芯片的 ROM 起始地址和大小的，右边的 IRAM1 是设置 STM32 芯片的 RAM 起始地址和大小的。其中 0X08000000 就是 STM32 内部 ROM 的起始地址，编译出来的指令肯定是要从 0X08000000 这个地址开始存放的。对于 STM32 来说 0X08000000 就是它的链接地址，图 8.4.1.2 中的这些.o 文件就是这个链接地址开始依次存放，最终生成一个可以下载的 hex 或者 bin 文件，我们可以打开.map 文件查看一下这些文件的链接地址，在 MDK 下打开一个工程的.map 文件方法如图 8.4.1.4 所示：

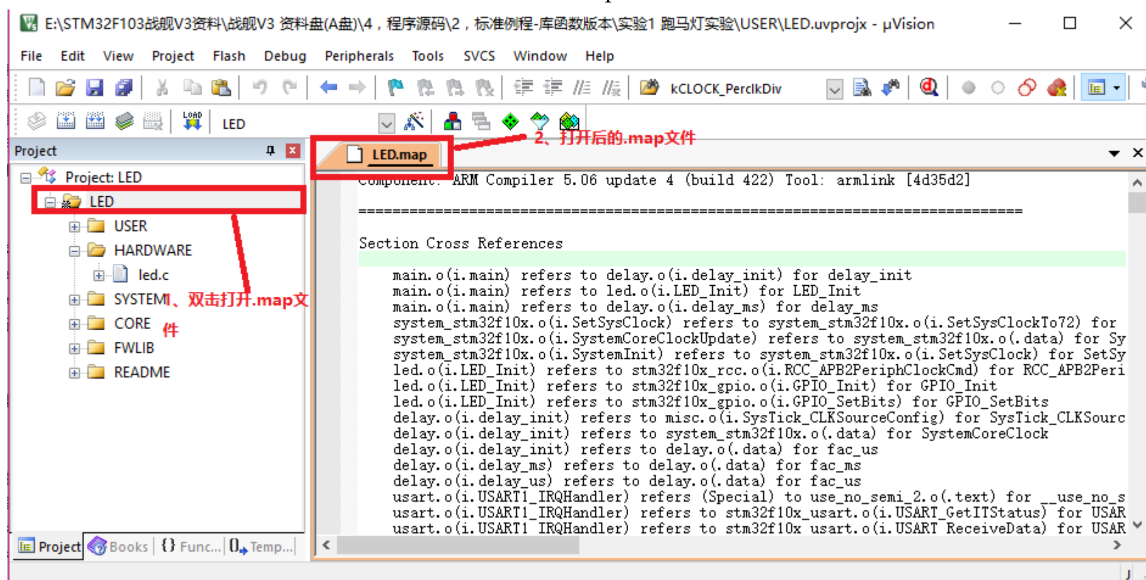


图 8.4.1.4 .map 文件打开方法

图 8.4.1.4 中的.map 文件就详细的描述了各个.o 文件都是链接到了什么地址，如图 8.4.1.5 所示：

```
LED.map

Memory Map of the image

  Image Entry point : 0x080001cd

  Load Region LR_1 (Base: 0x08000000, Size: 0x0000078c, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RO (Base: 0x08000000, Size: 0x0000076c, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000130   Data   RO         361     RESET               startup_stm32f10x_hd.o
    0x08000130   0x00000008   Code   RO         926   * !!!main             c_w.l(__main.o)
    0x08000138   0x00000034   Code   RO        1081     !!!scatter          c_w.l(__scatter.o)
    0x0800016c   0x0000001a   Code   RO        1083     !!handler_copy      c_w.l(__scatter_copy.o)
    0x08000186   0x00000002   PAD
    0x08000188   0x0000001c   Code   RO        1085     !!handler_zi        c_w.l(__scatter_zi.o)
    0x080001a4   0x00000002   Code   RO         955    .ARM.Collect$$libinit$$00000000  c_w.l(libinit.o)
    0x080001a6   0x00000000   Code   RO         962    .ARM.Collect$$libinit$$00000002  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         964    .ARM.Collect$$libinit$$00000004  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         967    .ARM.Collect$$libinit$$0000000A  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         969    .ARM.Collect$$libinit$$0000000C  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         971    .ARM.Collect$$libinit$$0000000E  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         974    .ARM.Collect$$libinit$$00000011  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         976    .ARM.Collect$$libinit$$00000013  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         978    .ARM.Collect$$libinit$$00000015  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         980    .ARM.Collect$$libinit$$00000017  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         982    .ARM.Collect$$libinit$$00000019  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         984    .ARM.Collect$$libinit$$0000001B  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         986    .ARM.Collect$$libinit$$0000001D  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         988    .ARM.Collect$$libinit$$0000001F  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         990    .ARM.Collect$$libinit$$00000021  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         992    .ARM.Collect$$libinit$$00000023  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         994    .ARM.Collect$$libinit$$00000025  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO         998    .ARM.Collect$$libinit$$0000002C  c_w.l(libinit2.o)
    0x080001a6   0x00000000   Code   RO        1000    .ARM.Collect$$libinit$$0000002E  c_w.l(libinit2.o)
```

图 8.4.1.5 STM32 镜像映射文件

从图 8.4.1.5 中就可以看出 STM32 的各个.o 文件所处的位置，起始位置是 0X08000000。由此可以得知，我们用 MDK 开发 STM32 的时候也是有链接的，只是这些工作 MDK 都帮我们全部做好了，我们不用关心而已。但是我们在 Linux 下用交叉编译器开发 ARM 的是时候就需要自己处理这些了。

因此我们现在需要做的就是确定一下本试验最终的可执行文件其运行起始地址，也就是链接地址。这里我们要区分“存储地址”和“运行地址”这两个概念，“存储地址”就是可执行文件存储在哪里，可执行文件的存储地址可以随意选择。“运行地址”就是代码运行的时候所处的地址，这个我们在链接的时候就已经确定好了，代码要运行，那就必须处于运行地址处，否则代码肯定运行出错。比如 I.MX6U 支持 SD 卡、EMMC、NAND 启动，因此代码可以存储到 SD 卡、EMMC 或者 NAND 中，但是要运行的话就必须将代码从 SD 卡、EMMC 或者 NAND 中拷贝到其运行地址(链接地址)处，“存储地址”和“运行地址”可以一样，比如 STM32 的存储起始地址和运行起始地址都是 0X08000000。

本教程所有的裸机例程都是烧写到 SD 卡中，上电以后 I.MX6U 的内部 boot rom 程序会将可执行文件拷贝到链接地址处，这个链接地址可以在 I.MX6U 的内部 128KB RAM 中(0X900000~0X91FFFF)，也可以在外部的 DDR 中。本教程所有裸机例程的链接地址都在 DDR 中，链接起始地址为 0X87800000。I.MX6U-ALPHA 开发板的 DDR 容量有两种：512MB 和 256MB，起始地址都为 0X80000000，只不过 512MB 的终止地址为 0X9FFFFFFF，而 256MB 容量的终止地址为 0X8FFFFFFF。之所以选择 0X87800000 这个地址是因为后面要讲的 Uboot 其链接地址就是 0X87800000，这样我们统一使用 0X87800000 这个链接地址，不容易记混。

确定了链接地址以后就可以使用 arm-linux-gnueabihf_ld 来将前面编译出来的 led.o 文件链接到 0X87800000 这个地址，使用如下命令：

```
arm-linux-gnueabihf-ld -Ttext 0X87800000 led.o -o led.elf
```

上述命令中-Ttext 就是指定链接地址，“-o”选项指定链接生成的 elf 文件名，这里我们命名为 led.elf。上述命令执行完以后就会在工程目录下多一个 led.elf 文件，如图 8.4.1.6 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.elf  led.o  led.s  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.6 链接生成 led.elf 文件

led.elf 文件也不是我们最终烧写到 SD 卡中的可执行文件，我们要烧写的.bin 文件，因此还需要将 led.elf 文件转换为.bin 文件，这里我们就需要用到 arm-linux-gnueabihf-objcopy 这个工具了。

3、arm-linux-gnueabihf-objcopy 格式转换

arm-linux-gnueabihf-objcopy 更像一个格式转换工具，我们需要用它将 led.elf 文件转换为 led.bin 文件，命令如下：

```
arm-linux-gnueabihf-objcopy -O binary -S -g led.elf led.bin
```

上述命令中，“-O”选项指定以什么格式输出，后面的“binary”表示以二进制格式输出，选项“-S”表示不要复制源文件中的重定位信息和符号信息，“-g”表示不复制源文件中的调试信息。上述命令执行完成以后，工程目录如图 8.4.1.7 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.bin  led.elf  led.o  led.s  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.7 生成最终的 led.bin 文件

至此我们终于等到了想要的东西——led.bin 文件。

4、arm-linux-gnueabihf-objdump 反汇编

大多数情况下我们都是用 C 语言写试验例程的，有时候需要查看其汇编代码来调试代码，因此就需要进行反汇编，一般可以将 elf 文件反汇编，比如如下命令：

```
arm-linux-gnueabihf-objdump -D led.elf  >  led.dis
```

上述代码中的“-D”选项表示反汇编所有的段，反汇编完成以后就会在当前目录下出现一个名为 led.dis 文件，如图 8.4.1.8 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.bin  led.dis  led.elf  led.o  led.s  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.8 反汇编生成 led.dis

可以打开 led.dis 文件看一下，看看是不是汇编代码，如图 8.4.1.9 所示：

```
 1
 2 led.elf:      文件格式 elf32-littlearm
 3
 4
 5 Disassembly of section .text:
 6
 7 87800000 <_start>:
 8 87800000:   e59f0068    ldr r0, [pc, #104]   ; 87800070 <loop+0x4>
 9 87800004:   e3e01000    mvn r1, #0
10 87800008:   e5801000    str r1, [r0]
11 8780000c:   e59f0060    ldr r0, [pc, #96]    ; 87800074 <loop+0x8>
12 87800010:   e5801000    str r1, [r0]
13 87800014:   e59f005c    ldr r0, [pc, #92]    ; 87800078 <loop+0xc>
14 87800018:   e5801000    str r1, [r0]
15 8780001c:   e59f0058    ldr r0, [pc, #88]    ; 8780007c <loop+0x10>
16 87800021:   e5801000    str r1, [r0]
17 87800024:   e59f0054    ldr r0, [pc, #84]    ; 87800080 <loop+0x14>
18 87800028:   e5801000    str r1, [r0]
19 8780002c:   e59f0050    ldr r0, [pc, #80]    ; 87800084 <loop+0x18>
20 87800030:   e5801000    str r1, [r0]
21 87800034:   e59f004c    ldr r0, [pc, #76]    ; 87800088 <loop+0x1c>
22 87800038:   e5801000    str r1, [r0]
23 8780003c:   e59f0048    ldr r0, [pc, #72]    ; 8780008c <loop+0x20>
```

图 8.4.1.9 反汇编文件

从图 8.4.1.9 可以看出 led.dis 里面是汇编代码，而且还可以看到内存分配情况。在 0X87800000 处就是全局标号_start，也就是程序开始的地方。通过 led.dis 这个反汇编文件可以明显的看出到我们的代码已经链接到了以 0X87800000 为起始地址的区域。

总结一下我们为了编译 ARM 开发板上运行的 led.o 这个文件使用了如下命令：

```
arm-linux-gnueabihf-gcc -g -c led.s -o led.o
arm-linux-gnueabihf-ld -Ttext 0X87800000 led.o -o led.elf
arm-linux-gnueabihf-objcopy -O binary -S -g led.elf led.bin
arm-linux-gnueabihf-objdump -D led.elf   >   led.dis
```

如果我们修改了 led.s 文件，那么就需要在重复一次上面的这些命令，太麻烦了，这个时候我们就可以使用第三章讲解的 Makefile 文件了。

8.4.2 创建 Makefile 文件

是用“touch”命令在工程根目录下创建一个名为“Makefile”的文件，如图 8.4.1.12 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ touch Makefile
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.bin  led.dis  led.elf  led.o  led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.12 创建 Makefile 文件

创建好 Makefile 文件以后就需要根据 Makefile 语法编写 Makefile 文件了，Makefile 基本语法我们已经在第三章讲解了，在 Makefile 中输入如下内容：

示例代码 8.4.2.1 Makefile 文件源码

```
led.bin:led.s
	arm-linux-gnueabihf-gcc -g -c led.s -o led.o
	arm-linux-gnueabihf-ld -Ttext 0X87800000 led.o -o led.elf
	arm-linux-gnueabihf-objcopy -O binary -S -g led.elf led.bin
	arm-linux-gnueabihf-objdump -D led.elf > led.dis
clean:
```

```
rm -rf *.o led.bin led.elf led.dis
```

创建好 Makefile 以后我们就只需要执行一次“make”命令即可完成编译，过程如图 8.4.1.13 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ make    //编译工程
arm-linux-gnueabihf-gcc -g -c led.s -o led.o
arm-linux-gnueabihf-ld -Ttext 0X87800000 led.o -o led.elf
arm-linux-gnueabihf-objcopy -O binary -S -g led.elf led.bin
arm-linux-gnueabihf-objdump -D led.elf > led.dis
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls    //查看编译结果
led.bin  led.dis  led.elf  led.o  led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.13 Makefile 执行过程

如果我们要清理工程的话执行“make clean”即可，如图 8.4.1.14 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
led.bin  led.dis  led.elf  led.o  led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ make clean  //清理工程
rm -rf *.o led.bin led.elf led.dis
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls    //查看清理后的工程
led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.1.14 make clean 清理工程

至此，有关代码编译、arm-linux-gnueabihf 交叉编译器的使用就到这里了，我们接下来讲解如何将 led.bin 烧写到 SD 卡中。

8.4.3 代码烧写

我们学习 STM32 等其他的单片机的时候，编译完代码以后可以直接通过 MDK 或者 IAR 下载到内部的 flash 中。但是 I.MX6U 虽然内部有 96K 的 ROM，但是这 96K 的 ROM 是 NXP 自己用的，不向用户开放。所以相当于说 I.MX6U 是没有内部 flash 的，但是我们的代码得有地方存放啊，为此，I.MX6U 支持从外置的 NOR Flash、NAND Flash、SD/EMMC、SPI NOR Flash 和 QSPI Flash 这些存储介质中启动，所以我们可以将代码烧写到这些存储介质中中。在这些存储介质中，除了 SD 卡以外，其他的一般都是焊接到了板子上的，我们没法直接烧写。但是 SD 卡是活动的，是可以从板子上插拔的，我们可以将 SD 卡插到电脑上，在电脑上使用软件将.bin 文件烧写到 SD 卡中，然后再插到板子上就可以了。其他的几种存储介质是我们量产的时候用到的，量产的时候代码就不可能放到 SD 卡里面了，毕竟 SD 是活动的，不牢固，而其他的都是焊接到板子上的，很牢固。

因此，我们在调试裸机和 Uboot 的时候是将代码下载到 SD 中，因为方便嘛，当调试完成以后量产的时候要将裸机或者 Uboot 烧写到 SPI NOF Flash、EMMC、NAND 等这些存储介质中的。那么，如何将我们前面编译出来的 led.bin 烧写到 SD 卡中呢？肯定有人会认为直接复制 led.bin 到 SD 卡中不就行了，错！编译出来的可执行文件是怎么存放到 SD 中的，存放的位置是什么？这个 NXP 是有详细规定的！我们必须按照 NXP 的规定来将代码烧写到 SD 卡中，否则代码是绝对运行不起来的。《IMX6UL 参考手册》的第 8 章“Chapter 8 System Boot”就是专门讲解 I.MX6U 启动的，我们下一章会详细的讲解 I.MX6U 启动方式的。

正点原子专门编写了一个软件来将编译出来的.bin 文件烧写到 SD 卡中，这个软件叫做“imxdownload”，软件我们已经放到了开发板光盘中，路径为：开发板光盘->5、开发工具->2、Ubuntu 下裸机烧写软件->imxdownload，imxdownlaod 只能在 Ubuntu 下使用，使用步骤如下：

1、将 imxdownload 拷贝到工程根目录下

我们要将 imxdownload 拷贝到工程根目录下，也就是和 led.bin 处于同一个文件夹下，要不然烧写会失败的，拷贝完成以后如图 8.4.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
imxdownload  led.bin  led.dis  led.elf  led.o  led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ a
```

图 8.4.3.1 拷贝 imxdownload 软件

2、给予 imxdownload 可执行权限

我们直接将软件 imxdownload 从 Windows 下复制到 Ubuntu 中以后，imxdownload 默认是没有可执行权限的。我们需要给予 imxdownload 可执行权限，使用命令“chmod”，命令如下：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ chmod 777 imxdownload  //给予可执行权限
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
imxdownload  led.bin  led.dis  led.elf  led.o  led.s  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.3.2 给予 imxdownload 可执行权限

通过对比图 8.4.3.1 和图 8.4.3.2 可以看到，当给予 imxdownload 可执行权限以后其名字变成了绿色的，如果没有可执行权限的话其名字颜色是白色的。所以在 Ubuntu 中我们可以初步的从文件名字的颜色判断其是否具有可执行权限。

3、确定要烧写的 SD 卡。

准备一张新的 SD(TF)卡，确保 SD 卡里面没有数据，因为我们在烧写代码的时候可能会格式话 SD 卡！！！

Ubuntu 下所有的设备文件都在目录“/dev”里面，所以插上 SD 卡以后也会出现在“/dev”里面，其中存储设备都是以“/dev/sd”开头的。我们要先看一下不插 SD 卡的时候电脑都有哪些存储设备，以防插入 SD 卡以后分不清谁是谁。输入如下所示命令：

```
ls /dev/sd*
```

当前电脑的存储文件如图 8.4.3.3 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.3.3 Ubuntu 当前存储文件

从图中可以看到当前电脑有/dev/sda、/dev/sda1、/dev/sda2 和/dev/sda5 这 5 个存储设备，使用读卡器将 SD 卡插到电脑，一定要确保 SD 卡是挂载到了 Ubuntu 系统中，而不是 Windows 下。SD 卡挂载到电脑以后，VMware 右下角会出现如图 8.4.3.4 所示图标：

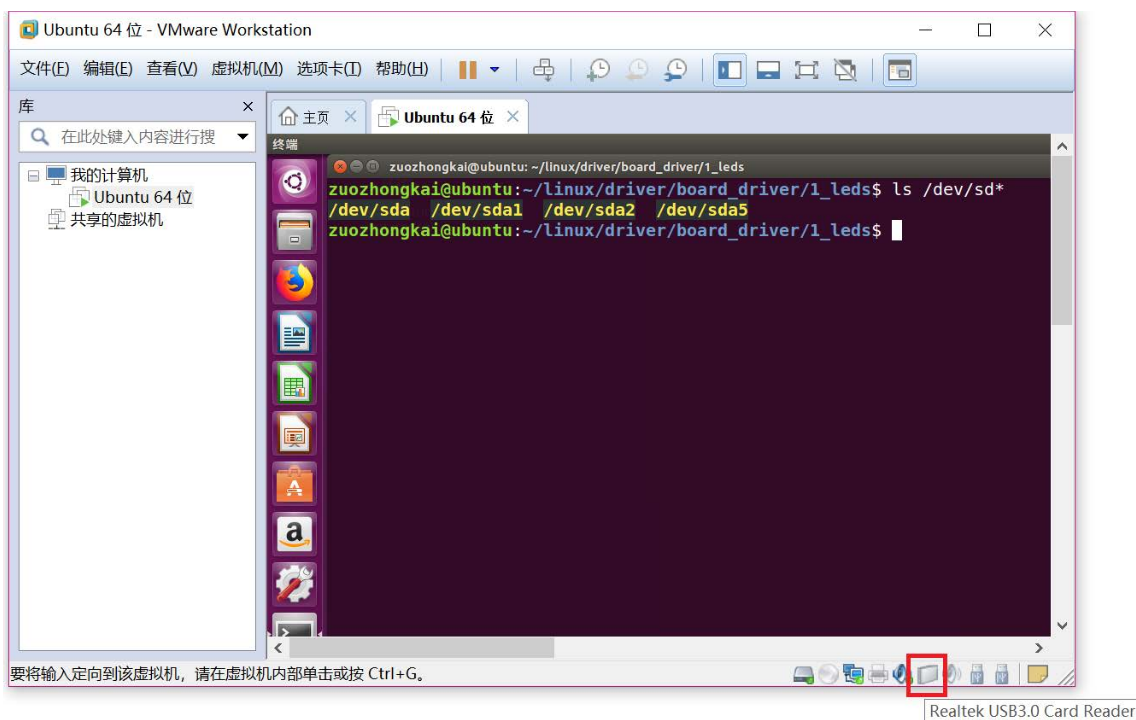


图 8.4.3.4 插上 SD 卡以后的提示

如图 8.4.3.4 所示，在 VMware 右小角有个图标看着像硬盘一样的图标：，这个图标就表示当前有存储设备插入，我们将鼠标放上去就会有提示当前设备名字，比如我这里提示“Realtek USB3.0 Card Reader”，这是我的读卡器的名字。如果是灰色的话就表示 SD 卡挂载到了 Windows 下，而不是 Ubuntu 上，所以我现在这个电脑的 SD 卡就是挂载到了 Windiows 下，我肯定要将其挂载到 Ubuntu 中，因为我要在 Ubuntu 下烧写代码。方法很简单，点击图标，点击以后如图 8.4.3.5 所示：

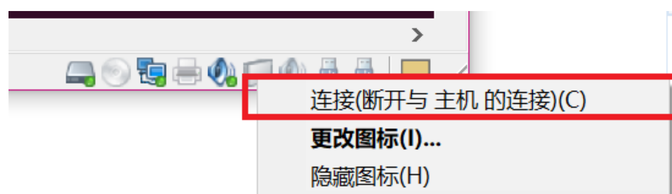


图 8.4.3.5 将 SD 卡连接到 Ubuntu 中

点击图 8.4.3.5 中的“连接(断开与主机的连接)(C)”，点击以后会弹出如图 8.4.3.6 所示提示界面：

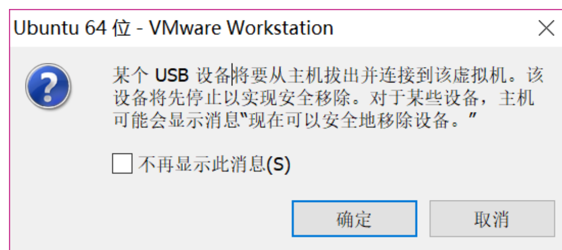


图 8.4.3.6 提示界面

图 8.4.3.6 提示你有个 USB 要从主机(Windows)拔出，插入虚拟机中，点击“确定”按钮即可。SD 卡出入到 Ubuntu 以后，图标就会变为，不是灰色的了。在输入命令“ls /dev/sd*”来查看当前 Ubuutu 下的存储设备，如图 8.4.3.7 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls /dev/sd*
/dev/sda  /dev/sda1  /dev/sda2  /dev/sda5  /dev/sdb  /dev/sdc  /dev/sdd  /dev/sdd1  /dev/sde  /dev/sdf
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.3.7 当前系统存储设备

从图 8.4.3.7 中可以看到，我的电脑多出了/dev/sdb、/dev/sdc、/dev/sdd、/dev/sdd1、/dev/sda 和/dev/sdf 这 6 个存储设备。这是因为我的读卡器是多合一读卡器，所以会多出来这么多，如果你用的单一读卡器那么应该只会出现一个或者两个。那这 6 个存储设备哪个才是我的 SD 卡呢？/dev/sdd 和/dev/sdd1 是我的 SD 卡，为什么呢？因为只有/dev/sdd 有个对应的/dev/sdd1，/dev/sdd 是我的 SD 卡，/dev/sdd1 是 SD 卡的第一个分区。如果你的 SD 卡有多个分区的话可能会出现/dev/sdd2、/dev/sdd3 等等。确定好 SD 卡以后我们就可以使用软件 imxdownload 向 SD 卡烧写 led.bin 文件了。

如果你的电脑没有找到 SD 卡的话，尝试重启一下 Ubuntu 操作！

4、向 SD 卡烧写 bin 文件

使用 imxdownload 向 SD 卡烧写 led.bin 文件，命令格式如下：

```
./imxdownload    <.bin file>    <SD Card>
```

其中.bin 就是要烧写的.bin 文件，SD Card 就是你要烧写的 SD 卡，比如我的电脑使用如下命令烧写 led.bin 到/dev/sdd 中：

```
./imxdownload led.bin /dev/sdd
```

烧写的过程中可能会让你输入密码，输入你的 Ubuntu 密码即可完成烧写，烧写过程如图 8.4.3.8 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ./imxdownload led.bin /dev/sdd
I.MX6UL bin download software
Edit by:zuozhongkai
Date:2018/8/9
Version:V1.0
file led.bin size = 160Bytes
Delete Old load.imx
Create New load.imx
Download load.imx to /dev/sdd  ......
[sudo] zuozhongkai 的密码：
记录了6+1 的读入
记录了6+1 的写出
3232 bytes (3.2 kB, 3.2 KiB) copied, 0.0160821 s, 201 kB/s
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.3.8 imxdownload 烧写过程

在图 8.4.3.8 中，烧写的最后一行会显示烧写大小、用时和速度，比如 led.bin 烧写到 SD 卡中的大小是 3.2KB，用时 0.0160821s，烧写速度是 201KB/s。注意这个烧写速度，如果这个烧写速度在几百 KB/s 以下那么就是正常烧写。

如果这个烧写速度大于几十 MB/s、甚至几百 MB/s 那么肯定是烧写失败了！

如果这个烧写速度大于几十 MB/s、甚至几百 MB/s 那么肯定是烧写失败了！

如果这个烧写速度大于几十 MB/s、甚至几百 MB/s 那么肯定是烧写失败了！

解决方法就是重新插拔 SD 卡，一般出现这种情况，重新插拔 SD 卡基本没啥用，只有重启 Ubuntu，至于原因，我也不清楚。

烧写完成以后会在当前工程目录下生成一个 load.imx 的文件，如图 8.4.3.9 所示：

```
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$ ls
imxdownload  led.bin  led.dis  led.elf  led.o  led.s  load.imx  Makefile  SI
zuozhongkai@ubuntu:~/linux/driver/board_driver/1_leds$
```

图 8.4.3.9 生成的 load.imx 文件

load.imx 这个文件就是软件 imxdownload 根据 NXP 官方启动方式介绍的内容，在 led.bin 文件前面添加了一些数据头以后生成的。最终烧写到 SD 卡里面的就是这个 load.imx 文件，而非 led.bin。至于具体添加了些什么内容，我们会在下一章讲解。

8.4.4 代码验证

代码已经烧写到了 SD 卡中了，接下来就是将 SD 卡插到开发板的 SD 卡槽中，然后设置拨码开关为 SD 卡启动，拨码开关设置如图 8.4.4.1 所示：

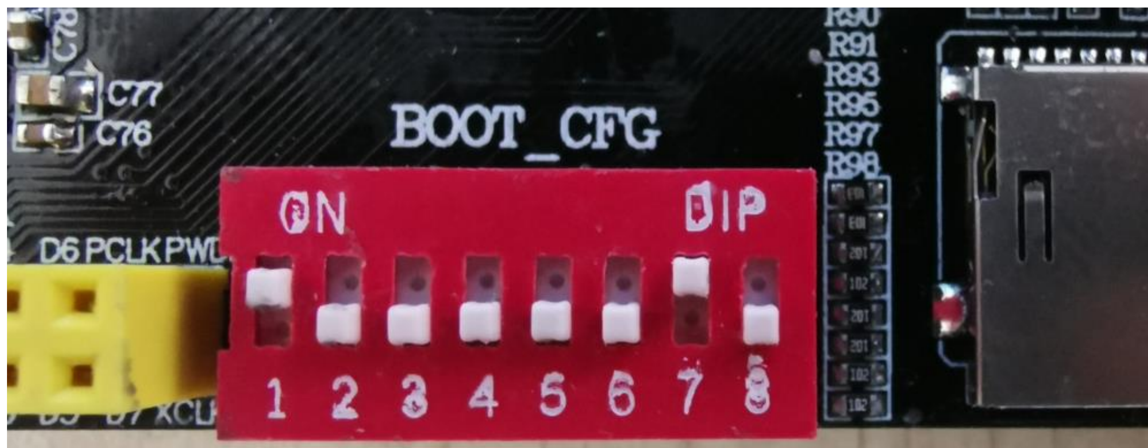


图 8.4.4.1 拨码开关 SD 卡启动设置

设置好以后按一下开发板的复位键，如果代码运行正常的话 LED0 就会被点亮，如图 8.4.4.2 所示：



图 8.4.4.2 LED0 点亮

如图 8.4.4.2 所示，LED0 被正常点亮，可能 LED0 之前会有一点微亮，那是因为 I.MX6U 的 IO 默认电平可能让 LED0 导通了，但是 IO 的默认配置内部可能有很大的电阻，所以电流就很小，导致 LED0 微亮。但是我们自己编写代码、配置好 IO 以后就不会有这个问题，LED0 就很亮了。

本小节我们详细的讲解了如何编译代码，并且如何将代码烧写进 SD 卡中进行测试。后续我们的所有裸机实验和 Uboot 实验都使用的这种方法进行代码的烧写和测试。

第九章 I.MX6U 启动方式详解

I.MX6U 支持多种启动方式以及启动设备，比如可以从 SD/EMMC、NAND Flash、QSPI Flash 等启动。用户可以根据实际情况，选择合适的启动设备。不同的启动方式其启动方式和启动要求也不一样，比如上一章中的从 SD 卡启动就需要在 bin 文件前面添加一个数据头，其它的启动设备也是需要这个数据头的。本章我们就来学习一下 I.MX6U 的启动方式，以及不同设备启动的要求。

9.1 启动方式选择

BOOT 的处理过程是发生在 I.MX6U 芯片上电以后，芯片会根据 BOOT_MODE[1:0]的设置来选择 BOOT 方式。BOOT_MODE[1:0]的值是可以改变的，有两种方式，一种是改写 eFUSE(熔丝)，一种是修改相应的 GPIO 高低电平。第一种修改 eFUSE 的方式只能修改一次，后面就不能在修改了，所以我们不使用。我们使用的是通过修改 BOOT_MODE[1:0]对应的 GPIO 高低电平来选择启动方式，所有的开发板都使用的这种方式，I.MX6U 有一个 BOOT_MODE1 引脚和 BOOT_MODE0 引脚，这两个引脚对应这 BOOT_MODE[1:0]。I.MX6U-ALPHA 开发板的这两个引脚原理图如图 9.1.1 所示：

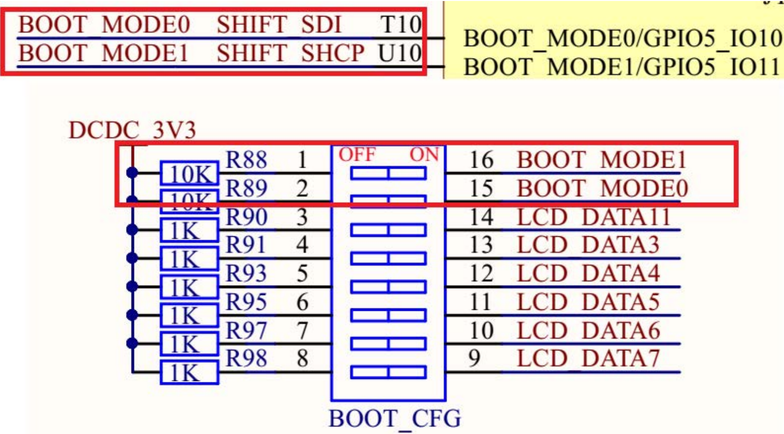


图 9.1.1 BOOT_MODE 原理图

其中 BOOT_MODE1 和 BOOT_MODE0 在芯片内部是有 100KΩ下拉电阻的，所以默认是 0。BOOT_MODE1 和 BOOT_MODE0 这两个引脚我们也接到了底板的拨码开关上，这样我们就可以通过拨码开关来控制 BOOT_MODE1 和 BOOT_MODE0 的高低电平。以 BOOT_MODE1 为例，当我们把 BOOT_CFG 的第一个开关拨到“ON”的时候，就相当于 BOOT_MODE1 引脚通过 R88 这个 10K 电阻接到了 3.3V 电源，芯片内部的 BOOT_MODE1 又是 100K 下拉电阻接地，因此此时 BOOT_MODE1 的电压就是 100/(10+100)*3.3V=3V，这是个高电平，BOOT_CFG 这个拨码开关的不管哪个位拨到“ON”就是高电平，拨到“OFF”就是低电平。

而 I.MX6U 有四个 BOOT 模式，这四个 BOOT 模式由 BOOT_MODE[1:0]来控制，也就是 BOOT_MODE1 和 BOOT_MODE0 这两 IO，BOOT 模式配置如表 9.1.1 所示：

BOOT_MODE[1:0]	BOOT 类型
00	从 FUSE 启动
01	串行下载
10	内部 BOOT 模式
11	保留

表 9.1.1 BOOT 类型

在表 9.1.1 中，我们用到的只有第二和第三种 BOOT 方式。

9.1.1 串行下载

当 BOOT_MODE1 为 0，BOOT_MODE0 为 1 的时候此模式使能，串行下载的意思就是可以通过 USB 或者 UART 将代码下载到板子上的外置存储设备中，我们可以使用 OTG1 这个 USB 口向开发板上的 SD/EMMC、NAND 等存储设备下载代码。我们需要将 BOOT_MODE1 拨到“OFF”，将 BOOT_MODE0 拨到“ON”。这个下载是需要用到 NXP 提供的一个软件，一般用

来最终量产的时候将代码烧写到外置存储设备中的，我们后面讲解如何使用。

9.1.2 内部 BOOT 模式

当 BOOT_MODE 为 1，BOOT_MODE0 为 0 的时候此模式使能，在此模式下，芯片会执行内部的 boot ROM 代码，这段 boot ROM 代码会进行硬件初始化(一部分外设)，然后从 boot 设备(就是存放代码的设备、比如 SD/EMMC、NAND)中将代码拷贝出来复制到指定的 RAM 中，一般是 DDR。

9.2 BOOT ROM 初始化内容

当我们设置 BOOT 模式为“内部 BOOT 模式”以后，I.MX6U 内部的 boot ROM 代码就会执行，这个 boot ROM 代码都会做什么处理呢？首先肯定是初始化时钟，boot ROM 设置的系统时钟如图 9.2.1 所示：

Clock	CCM signal	Source	Frequency (MHz) BT_FREQ=0	Frequency (MHz) BT_FREQ=1
ARM PLL	pll1_sw_clk		396	396
System PLL	pll2_sw_clk		528	528
USB PLL	pll3_sw_clk		480	480
AHB	ahb_clk_root	528 MHz PLL/PFD352	132	88
IPG	ipg_clk_root	528 MHz PLL/PFD352	66	44

图 9.2.1 boot ROM 系统时钟设置

在图 9.2.1 中 BT_FREQ 模式为 0，可以看到，boot ROM 会将 I.MX6U 的内核时钟设置为 396MHz，也就是主频为 396Mhz。System PLL=528Mhz，USB PLL=480MHz，AHB=132MHz，IPG=66MHz。关于 I.MX6U 的系统时钟，我们后面会详细讲解。

内部 boot ROM 为了加快执行速度会打开 MMU 和 Cache，下载镜像的时候 L1 ICache 会打开，验证镜像的时候 L1 DCache、L2 Cache 和 MMU 都会打开。一旦镜像验证完成，boot ROM 就会关闭 L1 DCache、L2 Cache 和 MMU。

中断向量偏移会被设置到 boot ROM 的起始位置，当 boot ROM 启动了用户代码以后就可以重新设置中断向量偏移了。一般是重新设置到我们用户代码的开始地方，关于中断的内容后面会详细讲解。

9.3 启动设备

当 BOOT_MODE 设置为内部 BOOT 模式以后，可以从一下设备中启动：

、接到 EIM 接口的 CS0 上的 16 位 NOR Flash。

、接到 EIM 接口的 CS0 上的 OneNAND Flash。

、接到 GPMI 接口上的 MLC/SLC NAND Flash，NAND Flash 页大小支持 2KByte、4KByte 和 8KByte，8 位宽。

、Quad SPI Flash。

、接到 USDHC 接口上的 SD/MMC/eSD/SDXC/eMMC 等设备。

、SPI 接口的 EEPROM。

这些启动设备如何选择呢？I.MX6U 同样提供了 eFUSE 和 GPIO 配置两种，eFUSE 就不讲解了。我们重点看如何通过 GPIO 来选择启动设备，因为所有的 I.MX6U 开发板都是通过 GPIO 来配置启动设备的。正如启动模式由 BOOT_MODE[1:0]来选择一样，启动设备是通过

BOOT_CFG1[7:0]、BOOT_CFG2[7:0]和 BOOT_CFG4[7:0]这 24 个配置 IO，这 24 个配置 IO 刚好对应着 LCD 的 24 根数据线 LCD_DATA0~LCDDATA23，当启动完成以后这 24 个 IO 就可以作为 LCD 的数据线使用。这 24 根线和 BOOT_MODE1、BOOT_MODE0 共同组成了 I.MX6U 的启动选择引脚，如图 9.3.1 所示：

Package Pin	Direction on reset	eFuse
BOOT_MODE1	Input	Boot Mode Selection
BOOT_MODE0	Input	
LCD1_DATA00	Input	BOOT_CFG1[0]
LCD1_DATA01	Input	BOOT_CFG1[1]
LCD1_DATA02	Input	BOOT_CFG1[2]
LCD1_DATA03	Input	BOOT_CFG1[3]
LCD1_DATA04	Input	BOOT_CFG1[4]
LCD1_DATA05	Input	BOOT_CFG1[5]
LCD1_DATA06	Input	BOOT_CFG1[6]
LCD1_DATA07	Input	BOOT_CFG1[7]
LCD1_DATA08	Input	BOOT_CFG2[0]
LCD1_DATA09	Input	BOOT_CFG2[1]
LCD1_DATA10	Input	BOOT_CFG2[2]
LCD1_DATA11	Input	BOOT_CFG2[3]
LCD1_DATA12	Input	BOOT_CFG2[4]
LCD1_DATA13	Input	BOOT_CFG2[5]
LCD1_DATA14	Input	BOOT_CFG2[6]
LCD1_DATA15	Input	BOOT_CFG2[7]
LCD1_DATA16	Input	BOOT_CFG4[0]
LCD1_DATA17	Input	BOOT_CFG4[1]
LCD1_DATA18	Input	BOOT_CFG4[2]
LCD1_DATA19	Input	BOOT_CFG4[3]
LCD1_DATA20	Input	BOOT_CFG4[4]
LCD1_DATA21	Input	BOOT_CFG4[5]
LCD1_DATA22	Input	BOOT_CFG4[6]
LCD1_DATA23	Input	BOOT_CFG4[7]

图 9.3.1 启动引脚

通过图 9.3.1 中的 26 个启动 IO 即可实现 I.MX6U 从不同的设备启动，BOOT_MODE1 和 BOOT_MODE0 已经讲过了。看到这 24 个 IO 是不是头大？调整这 24 个 IO 的高低电平得多复杂啊？其实不然，虽然有 24 个 IO，但是实际需要调整的只有那几个 IO，其它的 IO 全部下拉接地即可，也就是设置为 0。打开 I.MX6U-ALPHA 开发板的核心板原理图，这 24 个 IO 的默认设置如图 9.3.1 所示：

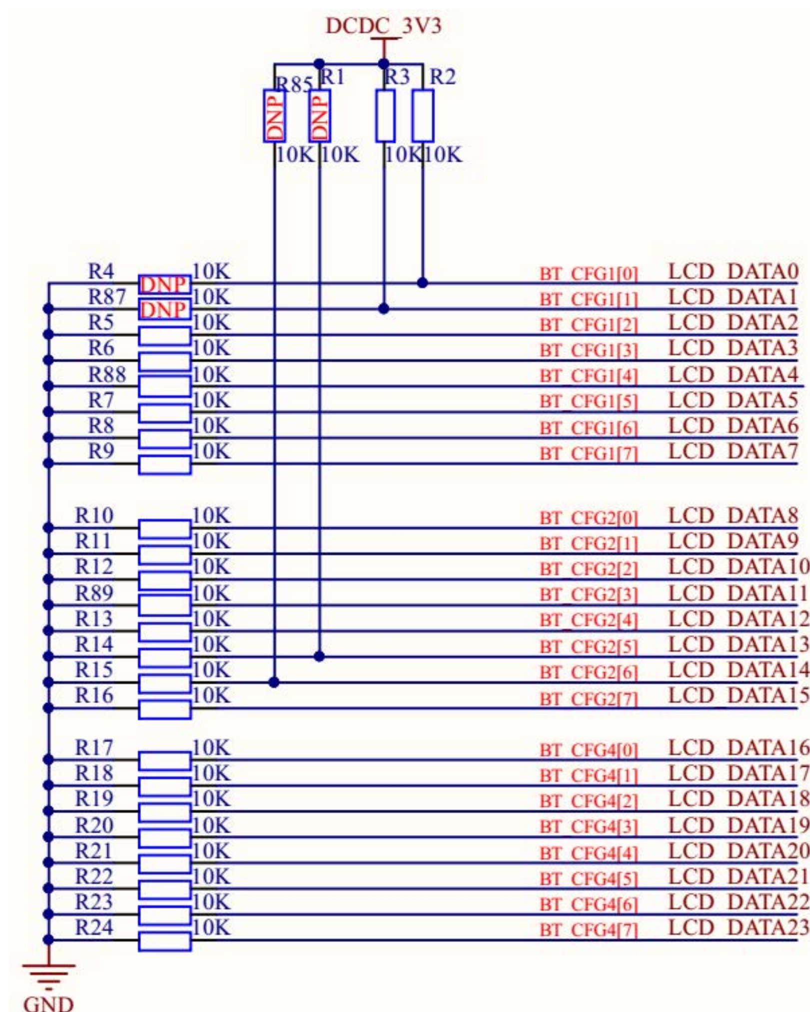


图 9.3.1 BOOT_CFG 默认设置

可以看出在图 9.3.1 中大部分的 IO 都接地了，只有几个 IO 接高，尤其是 BOOT_CFG4[7:0]这 8 个 IO 都 10K 电阻下拉接地，所以我们压根就不需要去关注 BOOT_CFG4[7:0]。我们需要重点关注的就只剩下了 BOOT_CFG2[7:0]和 BOOT_CFG1[7:0]这 16 个 IO。这 16 个配置 IO 含义在原理图的左侧已经贴出来了，如图 9.3.2 所示：

FUSE MAP <Default: QSPI BOOT> 0/1 0/1 0/1 1 0 0 0 0

TYPE	BOOT_CFG1[7]	BOOT_CFG1[6]	BOOT_CFG1[5]	BOOT_CFG1[4]	BOOT_CFG1[3]	BOOT_CFG1[2]	BOOT_CFG1[1]	BOOT_CFG1[0]
QSPI	0	0	0	1	Reserved	DDRSMP: "000" : Default "001-111"		
WEIM	0	0	0	0	Memory Type: 0 - NOR Flash 1 - OneNAND	Reserved	Reserved	Reserved
Serial-ROM	0	0	1	1	Reserved	Reserved	Reserved	Reserved
SD/eSD	0	1	0	Fast Boot: 0 - Regular 1 - Fast Boot	SD/SDXC Speed 00 - Normal/SDR12 01 - High/SDR25 10 - SDR50 11 - SDR104		SD Power Cycle Enable '0' - No power cycle '1' - Enabled via USDHC_RST pad (uSDHC3 & 4 only)	SD Loopback Clock Source Sel(for SDR50 and SDR104 only) '0' - through SD pad '1' - direct
MMC/eMMC	0	1	1	Fast Boot: 0 - Regular 1 - Fast Boot	SD/MMC Speed 0 - HighI 1- Normal	Fast Boot Acknowledge Disable: 0 - Boot Ack Enabled 1 - Boot Ack Disabled	SD Power Cycle Enable '0' - No power cycle '1' - Enabled via USDHC_RST pad (uSDHC3 & 4 only)	SD Loopback Clock Source Sel(for SDR50 and SDR104 only) '0' - through SD pad '1' - direct
NAND	1	BT_TOGGLEMODE	Pages In Block: 00 - 128 01 - 64 10 - 32 11 - 256		Nand Number Of Devices: 00 - 1 01 - 2 10 - 4 11 - Reserved		Nand_Row_address_bytes: 00 - 3 01 - 2 10 - 4 11 - 5	

TYPE	BOOT_CFG2[7]	BOOT_CFG2[6]	BOOT_CFG2[5]	BOOT_CFG2[4]	BOOT_CFG2[3]	BOOT_CFG2[2]	BOOT_CFG2[1]	BOOT_CFG2[0]
QSPI	Reserved	HSPHS: Half Speed Phase Selection 0 : select sampling at non-inverted clock 1: select sampling at inverted clock	HSDLY: Half Speed Delay selection 0 : one clock delay 1: two clock delay	FSPHS: Full Speed Phase Selection 0 : select sampling at non-inverted clock 1: select sampling at inverted clock	FSDLY: Full Speed Delay selection 0 : one clock delay 1: two clock delay	Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	Reserved	Reserved
WEIM	Muxing Scheme: 00 - A/D16 01 - A+DH 10 - A+DL 11- Reserved		OneNand Page Size: 00 - 1KB 01 - 2KB 10 - 4KB 11 - Reserved		Reserved	Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	Reserved	Reserved
Serial-ROM	Reserved	Reserved	Reserved	Reserved	Reserved	Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	Reserved	Reserved
SD/eSD	SD Calibration Step '00' - 1 TBD		Bus Width: 0 - 1-bit 1 - 4-bit	Port Select: 00 - eSDHC1 01 - eSDHC2 10 - Reserved 11 - Reserved		Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	SD1 VOLTAGE SELECTION 0 - 3.3V 1 - 1.8V	Reserved
MMC/eMMC	Bus Width: 000 - 1-bit 001 - 4-bit 010 - 8-bit 101 - 4-bit DDR (MMC 4.4) 110 - 8-bit DDR (MMC 4.4) Else - reserved.			Port Select: 00 - eSDHC1 01 - eSDHC2 10 - Reserved 11 - Reserved		Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	SD1 VOLTAGE SELECTION 0 - 3.3V 1 - 1.8V	Reserved
NAND	Toggle Mode 33MHz Preamble Delay, Read Latency: '000' - 16 GPMICLK cycles. '001' - 1 GPMICLK cycles. '010' - 2 GPMICLK cycles. '011' - 3 GPMICLK cycles. '100' - 4 GPMICLK cycles. '101' - 5 GPMICLK cycles. '110' - 6 GPMICLK cycles. '111' - 7 GPMICLK cycles.			BOOT_SEARCH_COUNT: 00 - 2 01 - 2 10 - 4 11 - 8		Boot Frequencies (ARM/DDR) 0 - 500 / 400 MHz 1 - 250 / 200 MHz	Reset Time '0' - 12ms '1' - 22ms (LBA Nand)	Reserved

图 9.3.2 BOOT_CFG 引脚含义

图 9.3.2 看着是不是也很头大，BOOT_CFG1[7:0]和 BOOT_CFG2[7:0]这 16 个 IO 还能不能在减少呢？可以！打开 I.MX6U-ALPHA 开发板的底板原理图，底板上启动设备选择拨码开关原理图如图 9.3.3 所示：

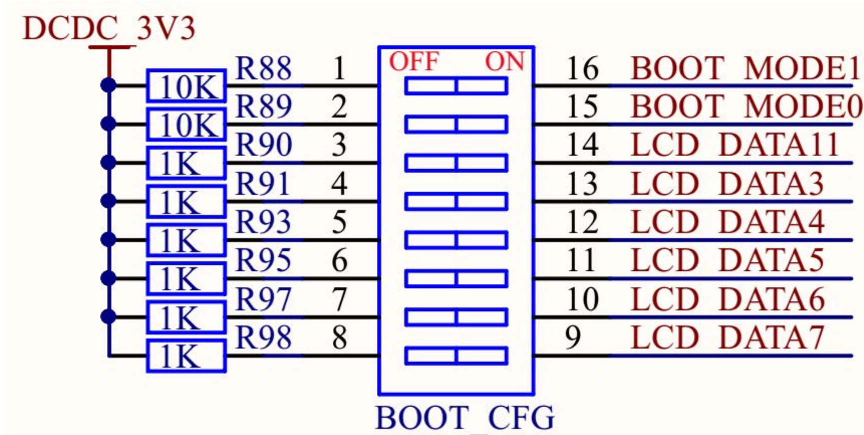


图 9.3.3 BOOT 选择拨码开关

在图 9.3.3 中，除了 BOOT_MODE1 和 BOOT_MODE0 必须引出来，LCD_DATA3~LCDDATA7、LCD_DATA11 这 6 个 IO 也被引出来了，可以通过拨码开关来设置其对应的高低电平，拨码开关拨到“ON”就是 1，拨到“OFF”就是 0。其中 LCD_DATA11 就是 BOOT_CFG2[3]，LCD_DATA3~LCD_DATA7 就是 BOOT_CFG1[3]~BOOT_CFG1[7]，这 6 个 IO 的配置含义如表 9.3.1 所示：

<table>
<tr><th>BOOT_CFG 引脚</th><th>对应 LCD 引脚</th><th>含义</th></tr>
<tr><td>BOOT_CFG2[3]</td><td>LCD_DATA11</td><td>为 0 时从 SDHC1 上的 SD/EMMC 启动，为 1 时从 SDHC2 上的 SD/EMMC 启动。</td></tr>
<tr><td>BOOT_CFG1[3]</td><td>LCD_DATA3</td><td>当从 SD/EMMC 启动的时候设置启动速度，当从 NAND 启动的话设置 NAND 数量。</td></tr>
<tr><td>BOOT_CFG1[4]</td><td>LCD_DATA4</td><td rowspan="5">BOOT_CFG1[7:4]：
0000 NOR/OneNAND(EIM)启动。
0001 QSPI 启动。
0011 SPI 启动。
010x SD/eSD/SDXC 启动。
011x MMC/eMMC 启动。
1xxx NAND Flash 启动。</td></tr>
<tr><td>BOOT_CFG1[5]</td><td>LCD_DATA5</td></tr>
<tr><td>BOOT_CFG1[6]</td><td>LCD_DATA6</td></tr>
<tr><td>BOOT_CFG1[7]</td><td>LCD_DATA7</td></tr>
</table>

表 9.3.1 BOOT IO 含义

根据表 9.3.1 中的 BOOT IO 含义，I.MX6U-ALPHA 开发板从 SD 卡、EMMC、NAND 启动的时候拨码开关各个位设置方式如表 9.3.2 所示：

1	2	3	4	5	6	7	8	启动设备
0	1	x	x	x	x	x	x	串行下载，可以通过 USB 烧写镜像文件。
1	0	0	0	0	0	1	0	SD 卡启动。
1	0	1	0	0	1	1	0	EMMC 启动。
1	0	0	0	1	0	0	1	NAND FLASH 启动。

表 9.3.2 I.MX6U-ALPHA 开发板启动设置

我们在“第八章 汇编 LED 灯试验”中，最终的可执行文件 led.bin 烧写到了 SD 卡里面，然后开发板从 SD 卡启动，其拨码开关就是根据表 9.3.2 来设置的，通过上面的讲解就知道为什么拨码开关要这么设置了。

9.4 镜像烧写

注意！本小节会分析 bin 文件添加的头部信息，但是在笔者写本教程的时候关于 I.MX 系列 SOC 头部信息的资料很少，基本只能参考 NXP 官方资料，而官方资料有些地方讲解的又不是很详细。所以本节有部分内容是笔者根据 NXP 官方 u-boo.imx 文件的头部信息反推出来的，因此难免有错误的地方，还望大家谅解！如有发现错误之处，欢迎大家在 www.openedv.com 论坛上留言。

前面我们设置好 BOOT 以后就能从指定的设备启动了，但是你的设备里面得有代码啊，在第八章中我们使用 imxdownload 这个软件将 led.bin 烧写到了 SD 卡中。imxdownload 会在 led.bin 前面添加一些头信息，重新生成一个叫做 load.imx 的文件，最终实际烧写的是 laod.imx。那么肯定就有人问：imxdownload 究竟做了什么？load.imx 和 led.bin 究竟是什么关系？本节我们就来详细的讲解一下 imxdownload 是如何将 led.bin 打包成 load.imx 的。

学习 STM32 的时候我们可以直接将编译生成的.bin 文件烧写到 STM32 内部 flash 里面，但是 I.MX6U 不能直接烧写编译生成的.bin 文件，我们需要在.bin 文件前面添加一些头信息构成满足 I.MX6U 需求的最终可烧写文件，I.MX6U 的最终可烧写文件组成如下：

、Image vector table，简称 IVT，IVT 里面包含了一系列的地址信息，这些地址信息在 ROM 中按照固定的地址存放着。

、Boot data，启动数据，包含了镜像要拷贝到哪个地址，拷贝的大小是多少等等。

、Device configuration data，简称 DCD，设备配置信息，重点是 DDR3 的初始化配置。

、用户代码可执行文件，比如 led.bin。

可以看出最终烧写到 I.MX6U 中的程序其组成为：IVT+Boot data+DCD+.bin。所以第八章中的 imxdownload 所生成的 laod.imx 就是在 led.bin 前面加上 IVT+Boot data+DCD。内部 Boot ROM 会将 load.imx 拷贝到 DDR 中，用户代码是要一定要从 0X87800000 这个地方开始的，因为链接地址为 0X87800000，laod.imx 在用户代码前面又有 3KByte 的 IVT+Boot Data+DCD 数据，下面会讲为什么是 3KByte，因此 load.imx 在 DDR 中的起始地址就是 0X87800000-3072=0X877FF400。

9.4.1 IVT 和 Boot Data 数据

load.imx 最前面的就是 IVT 和 Boot Data，IVT 包含了镜像程序的入口点、指向 DCD 的指针和一些用作其它用途的指针。内部 Boot ROM 要求 IVT 应该放到指定的位置，不同的启动设备位置不同，而 IVT 在整个 load.imx 的最前面，其实就相当于要求 load.imx 在烧写的时候应该烧写到存储设备的指定位置去。整个位置都是相对于存储设备的起始地址的偏移，如图 9.4.1.1 所示：

Boot Device Type	Image Vector Table Offset	Initial Load Region Size
NOR	4 Kbyte = 0x1000 bytes	Entire Image Size
OneNAND	256 bytes = 0x100 bytes	1 Kbyte
SD/MMC/eSD/eMMC/SDXC	1 Kbyte = 0x400 bytes	4 Kbyte
SPI EEPROM	1 Kbyte = 0x400 bytes	4 Kbyte

图 9.4.1.1 IVT 偏移

以 SD/EMMC 为例，IVT 偏移为 1Kbyte，IVT+Boot data+DCD 的总大小为 4KByte-1KByte=3KByte。假如 SD/EMMC 每个扇区为 512 字节，那么 load.imx 应该从第三个扇区开始烧写，前两个扇区要留出来。load.imx 从第 3KByte 开始才是真正的.bin 文件。那么 IVT 里面究竟存放着什么东西呢？IVT 里面存放的内容如图 9.4.1.2 所示：

header
entry: Absolute address of the first instruction to execute from the image
reserved1: Reserved and should be zero
dcd: Absolute address of the image DCD. The DCD is optional so this field may be set to NULL if no DCD is required. See Device Configuration Data (DCD) for further details on DCD.
boot data: Absolute address of the Boot Data
self: Absolute address of the IVT. Used internally by the ROM
csf: Absolute address of Command Sequence File (CSF) used by the HAB library. See High Assurance Boot (HAB) for details on secure boot using HAB. This field must be set to NULL when not performing a secure boot
reserved2: Reserved and should be zero

图 9.4.1.2 IVT 格式

从图 9.4.1.2 可以看到，第一个存放的就是 header(头)，header 格式如图 9.4.1.3 所示：

Tag	Length	Version

图 9.4.1.3 IVT header 格式

在图 9.4.1.3 中，Tag 为一个字节长度，固定为 0XD1，Length 是两个字节，保存着 IVT 长度，为大端格式，也就是高字节保存在低内存中。最后的 Version 是一个字节，为 0X40 或者

0X41。

Boot Data 的数据格式如图 9.4.1.4 所示：

start	Absolute address of the image
length	Size of the program image
plugin	Plugin flag (see Plugin Image)

图 9.4.1.4 Boot Data 数据格式

实际情况是不是这样的呢？我们用 winhex 软件打开 load.imx 一看便知，winhex 可以直接查看一个文件的二进制格式数据，winhex 软件我们已经放到了开发板光盘中，路径为：开发板光盘->3、软件-> winhexv19.7.zip，大家自行安装。用 winhex 打开以后的 load.imxd 如图 9.4.1.4 所示：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00000000	D1	00	20	40	00	00	80	87	00	00	00	00	2C	F4	7F	87	20	F4	7F	87	00	F4	7F	87	00	00	00	00	00	00	00	00
00000020	00	F0	7F	87	00	00	20	00	00	00	00	00	D2	01	E8	40	CC	01	E4	04	02	0C	40	68	FF	FF	FF	FF	02	0C	40	6C
00000040	FF	FF	FF	FF	02	0C	40	70	FF	FF	FF	FF	02	0C	40	74	FF	FF	FF	FF	02	0C	40	78	FF	FF	FF	FF	02	0C	40	7C
00000060	FF	FF	FF	FF	02	0C	40	80	FF	FF	FF	FF	02	0E	04	B4	00	0C	00	00	02	0E	04	AC	00	00	00	00	02	0E	02	7C
00000080	00	00	00	30	02	0E	02	50	00	00	00	30	02	0E	02	4C	00	00	00	30	02	0E	04	90	00	00	00	30	02	0E	02	88
000000A0	00	0C	00	30	02	0E	02	70	00	00	00	00	02	0E	02	60	00	00	00	30	02	0E	02	64	00	00	00	30	02	0E	04	A0
000000C0	00	00	00	30	02	0E	04	94	00	02	00	00	02	0E	02	80	00	00	00	30	02	0E	02	84	00	00	00	30	02	0E	04	B0
000000E0	00	02	00	00	02	0E	04	98	00	00	00	30	02	0E	04	A4	00	00	00	30	02	0E	02	44	00	00	00	30	02	0E	02	48
00000100	00	00	00	30	02	1B	00	1C	00	00	80	00	02	1B	08	00	A1	39	00	03	02	1B	08	0C	00	03	00	0B	02	1B	08	3C
00000120	01	48	01	44	02	1B	08	48	40	40	2C	30	02	1B	08	50	40	40	3E	34	02	1B	08	1C	33	33	33	33	02	1B	08	20
00000140	33	33	33	33	02	1B	08	2C	F3	33	33	33	02	1B	08	30	F3	33	33	33	02	1B	08	C0	00	94	40	09	02	1B	08	B8
00000160	00	00	08	00	02	1B	00	04	00	02	00	2D	02	1B	00	08	1B	33	30	30	02	1B	00	0C	67	6B	52	F3	02	1B	00	10
00000180	B6	6D	0B	63	02	1B	00	14	01	FF	00	DB	02	1B	00	18	00	20	17	40	02	1B	00	1C	00	00	80	00	02	1B	00	2C
000001A0	00	00	26	D2	02	1B	00	30	00	6B	10	23	02	1B	00	40	00	00	00	4F	02	1B	00	00	84	18	00	00	02	1B	08	90
000001C0	00	40	00	00	02	1B	00	1C	02	00	80	32	02	1B	00	1C	00	00	80	33	02	1B	00	1C	00	04	80	31	02	1B	00	1C
000001E0	15	20	80	30	02	1B	00	1C	04	00	80	40	02	1B	00	20	00	00	08	00	02	1B	08	18	00	00	02	27	02	1B	00	04
00000200	00	02	55	2D	02	1B	04	04	00	01	10	06	02	1B	00	1C	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000220	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

图 9.4.1.4 load.imx 部分内容

图 9.4.1.4 是我们截取的 load.imx 的一部分内容，从地址 0X00000000~0X000025F，共 608 个字节的数据。我们将前 44 个字节的数据按照 4 个字节一组组合在一起就是：0X402000D1、0X87800000、0X00000000、0X877FF42C、0X877FF420、0X877FF400、0X00000000、0X00000000、0X877FF000、0X00200000、0X00000000。这 44 个字节的数据就是 IVT 和 Boot Data 数据，按照图 9.4.1.2 和图 9.4.1.4 所示的 IVT 和 Boot Data 所示的格式对应起来如表 9.4.1.1 所示：

IVT		
IVT 结构	**数据**	**描述**
header	0X402000D1	根据图 9.4.1.3 的 header 格式，第一个字节 Tag 为 0XD1，第二三这两个字节为 IVT 大小，为大端模式，所以 IVT 大小为 0X20=32 字节。第四个字节为 0X40。完全符合图 9.4.1.3 中的格式。
entry	0X87800000	入口地址，也就是镜像第一行指令所在的位置。0X87800000 就是我们的链接地址。
reserved1	0X00000000	未使用，保留。
dcd	0X877FF42C	DCD 地址，镜像地址为 0X87800000，IVT+Boot Data+DCD 整个大小为 3KByte。因此 load.imx 的起始地址就是 0X87800000-0XC00=0X877FF400。因此 DCD 起始地址相对于 load.imx 起始地址的偏移就是 0X877FF42C-0X877FF400=0X2C，也就是说从图 9.4.1.4 中的 0X2C 这个地址开始就是 DCD 数据了。

boot data	0X877FF420	boot 地址，header 里面已经设置了 IVT 大小是 32 个字节，所以 boot data 的地址就是 0X877FF400+32=0X877FF420。
self	0X877FF400	IVT 复制到 DDR 中以后的首地址。
csf	0X00000000	CSF 地址。
reserved2	0X00000000	保留，未使用。
Boot Data		
Boot Data 结构	数据	描述
start	0X877FF000	整个 load.imx 的起始地址，包括前面 1KByte 的地址偏移。
length	0X00200000	镜像大小，这里设置 2MByte。镜像大小不能超过 2MByte。
plugin	0X00000000	插件。

表 9.4.1.1 load.imx 结构分析

在表 9.4.1.1 中，我们详细的列出了 load.imx 的 IVT+Boot Data 每 32 位数据所代表的意义。这些数据都是由 imxdownload 这个软件添加进去的。

9.4.2 DCD 数据

复位以后，I.MX6U 片内的所有寄存器都会复位为默认值，但是这些默认值往往不是我们想要的值，而且有些外设我们必须在使用之前初始化它。为此 I.MX6U 提出了一个 DCD(Device Config Data)的概念，和 IVT、Boot Data 一样，DCD 也是添加到 load.imx 里面的，紧跟在 IVT 和 Boot Data 后面，IVT 里面也指定了 DCD 的位置。DCD 其实就是 I.MX6U 寄存器地址和对应的配置信息集合，Boot ROM 会使用这些寄存器地址和配置集合来初始化相应的寄存器，比如开启某些外设的时钟、初始化 DDR 等等。DCD 区域不能超过 1768Byte，DCD 区域结构如图 9.4.2.1 所示：

Header
[CMD]
[CMD]
...

图 9.4.2.1 DCD 区域结构

DCD 的 header 和 IVT 的 header 类似，结构如图 9.4.2.2 所示：

Tag	Length	Version

图 9.4.2.2 DCD 的 header 结构

其中 Tag 是单字节，固定为 0XD2，Length 为两个字节，表示 DCD 区域的大小，包含 header，同样是大端模式，Version 是单字节，固定为 0X40 或者 0X41。

图 9.4.2.1 中的 CMD 就是要初始化的寄存器地址和相应的寄存器值，结构如图 9.4.2.3 所示：

Tag	Length	Parameter
Address		
Value/Mask		
[Address]		
[Value/Mask]		
...		
[Address]		
[Value/Mask]		

图 9.4.2.3 DCD CMD 命令格式

图 9.4.2.3 中 Tag 为一个字节，固定为 0XCC。Length 是两个字节，包含写如的命令数据的长度，包含 header，同样是大端模式。Parameter 为一个字节，这个字节的每个位含义如图 9.4.2.4 所示：

7	6	5	4	3	2	1	0
flags					bytes		

图 9.4.2.4 Parameter 结构

图 9.4.2.4 中的 bytes 表示是目标位置宽度，单位为 byte，可以选择 1、2、和 4 字节。flags 是命令控制标志位。

图 9.4.2.3 中的 Address 和 Vlalue/Mask 就是要初始化的寄存器地址和相应的寄存器值，注意采用的是大端模式！DCD 结构就分析到这里，在分析 IVT 的时候我们就已经说过了，DCD 数据是从图 9.4.1.4 的 0X2C 地址开始的。根据我们分析的 DCD 结构可以得到 load.imx 的 DCD 数据如表 9.4.2.1 所示：

DCD 结构	数据	描述
header	0X40E801D2	根据图 9.4.2.2 的 header 格式，第一个字节 Tag 为 0XD2，第二和三这两个字节为 DCD 大小，为大端模式，所以 DCD 大小为 0X01E8=488 字节。第四个字节为 0X40。完全符合图 9.4.2.2 中的格式。
Write Data Command	0X04E401CC	根据图 9.4.2.3，第一个为 Tag，固定为 0XCC，第二和三这两个字节是大端模式的命令总长度，为 0X01E4=484 个字节。第四个字节是 Parameter，为 0X04，表示目标位置宽度为 4 个字节。
Address	0X020C4068	寄存器 CCGR0 地址
Value	0XFFFFFFFF	要写入寄存器 CCGR0 的值，表示打开 CCGR0 控制的所有外设时钟。
……	……	CCGR1~CCGR5 这些寄存器的地址和值。
Address	0X020C4080	寄存器 CCGR6 地址
Value	0XFFFFFFFF	要写入寄存器 CCGR6 的值，表示打开 CCGR6 控制的所有外设时钟。
Address	0X020E04B4	寄存器 IOMUXC_SW_PAD_CTL_GRP_DDR_TYPE 寄存器地址。
Value	0X000C0000	设置 DDR 的所有 IO 为 DDR3 模式。
Address	0X020E04AC	寄存器 IOMUXC_SW_PAD_CTL_GRP_DDRPKE 地址。
Value	0X00000000	所有 DDR 引脚关闭 Pull/Keeper 功能。
Address	0X020E027C	寄存器 IOMUXC_SW_PAD_CTL_PAD_DRAM_SDCLK0_P
Value	0X00000030	DRAM_SDCLK0_P 引脚为 R0/6。

……	……	全部是 DDR 引脚设置
Address	0X020E0248	寄存器 IOMUXC_SW_PAD_CTL_PAD_DRAM_DQM1
Value	0X00000030	DRAM_DQM1 引脚驱动能力为 R0/6
Address	0X021B001C	MMDC_MDSCR 寄存器
Value	0X00008000	MMDC_MDSCR 寄存器值
……	……	MMDC 相关寄存器地址及其寄存器值。
Address	0X021B0404	MMDC_MAPSR 寄存器
Value	0X00011006	MMDC_MAPSR 寄存器配置值
Address	0X021B001C	MMDC_MDSCR 寄存器
Value	0X00000000	MMDC_MDSCR 寄存器清零

表 9.4.2.1 DCD 数据结构

从图 9.4.2.1 中可以看出，DCD 里面的初始化配置主要包括三方面：

①、设置 CCGR0~CCGR6 这 7 个外设时钟使能寄存器，默认打开所有的外设时钟。

②、配置 DDR3 所用的所有 IO。

③、配置 MMDC 控制器，初始化 DDR3。

I.MX6U 的启动过程我们就讲解到这里，本章我们详细的讲解了 I.MX6U 的启动模式、启动设备类型和镜像烧写过程。总结一下，我们编译出来的.bin 文件不能直接烧写到 SD 卡中，需要在.bin 文件前面加上 IVT、Boot Data 和 DCD 这三个数据块。这三个数据块是有指定格式的，我们必须按照格式填写，然后将其放到.bin 文件前面，最终合成的才是可以直接烧写到 SD 卡中的文件。

第十章 C 语言版 LED 灯实验

第八章我们讲解了如何用汇编语言编写 LED 灯实验，但是实际开发过程中汇编用的很少，大部分都是 C 语言开发，汇编只是用来完成 C 语言环境的初始化。本章我们就来学习如何用汇编来完成 C 语言环境的初始化工作，然后从汇编跳转到 C 语言代码里面去。

10.1 C 语言版 LED 灯简介

第八章的汇编 LED 灯实验中，我们讲解了如何使用汇编来编写 LED 灯驱动，实际工作中是很少用到汇编去写嵌入式驱动的，毕竟汇编太难，而且写出来也不好理解，大部分情况下都是使用 C 语言去编写的。只是在开始部分用汇编来初始化一下 C 语言环境，比如初始化 DDR、设置堆栈指针 SP 等等，当这些工作都做完以后就可以进入 C 语言环境，也就是运行 C 语言代码，一般都是进入 main 函数。所以我们有两部分文件要做：

①、汇编文件

汇编文件只是用来完成 C 语言环境搭建。

②、C 语言文件

C 语言文件就是完成我们的业务层代码的，其实就是我们实际例程要完成的功能。

其实 STM32 也是这样的，只是我们在开发 STM32 的时候没有想到这一点，以 STM32F103 为例，其启动文件 startup_stm32f10x_hd.s 这个汇编文件就是完成 C 语言环境搭建的，当然还有一些其他的处理，比如中断向量表等等。当 startup_stm32f10x_hd.s 把 C 语言环境初始化完成以后就会进入 C 语言环境。

10.2 硬件原理分析

本章使用到的硬件资源和第八章一样，就是一个 LED0。

10.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程->2_ledc。

新建 VScode 工程，工程名字为“ledc”，新建三个文件：start.S、main.c 和 mian.h。其中 start.S 是汇编文件，main.c 和 main.h 是 C 语言相关文件。

10.3.1 汇编部分实验程序编写

在 STM32 中，启动文件 startup_stm32f10x_hd.s 就是完成 C 语言环境搭建的，当然还有一些其他的处理，比如中断向量表等等。startup_stm32f10x_hd.s 中堆栈初始化代码如下所示：

示例代码 10.3.1.1 STM32 启动文件堆栈初始化代码

```
1  Stack_Size      EQU     0x00000400
2
3                  AREA    STACK, NOINIT, READWRITE, ALIGN=3
4  Stack_Mem       SPACE   Stack_Size
5  __initial_sp
6
7  ; <h> Heap Configuration
8  ;   <o>  Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
9  ; </h>
10
11 Heap_Size       EQU     0x00000200
12
13                 AREA    HEAP, NOINIT, READWRITE, ALIGN=3
14 __heap_base
15 Heap_Mem        SPACE   Heap_Size
```

```
16 __heap_limit
17 ******************省略掉部分代码**********************
18 Reset_Handler   PROC
19                 EXPORT  Reset_Handler             [WEAK]
20                 IMPORT  __main
21                 IMPORT  SystemInit
22                 LDR     R0, =SystemInit
23                 BLX     R0
24                 LDR     R0, =__main
25                 BX      R0
26                 ENDP
```

第 1 行代码就是设置栈大小，这里是设置为 0X400=1024 字节。

第 5 行的__initial_sp 就是初始化 SP 指针。

第 11 行是设置堆大小。

第 18 行是复位中断服务函数，STM32 复位完成以后会执行此中断服务函数。

第 22 行调用 SystemInit()函数来完成其他初始化工作。

第 24 行调用__main，__main 是库函数，其会调用 main()函数。

I.MX6U 的汇编部分代码和 STM32 的启动文件 startup_stm32f10x_hd.s 基本类似的，只是本实验我们不考虑中断向量表，只考虑初始化 C 环境即可。在前面创建的 start.s 中输入如下代码：

示例代码 10.3.1.2 start.s 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : start.s
作者     : 左忠凯
版本     : V1.0
描述     : I.MX6U-ALPHA/I.MX6ULL 开发板启动文件，完成 C 环境初始化，
           C 环境初始化完成以后跳转到 C 代码。
其他     : 无
日志     : 初版 2019/1/3 左忠凯修改
**************************************************************/

1  .global _start       /* 全局标号 */
2
3  /*
4   * 描述：  _start 函数，程序从此函数开始执行，此函数主要功能是设置 C
5   *         运行环境。
6   */
7  _start:
8
9      /* 进入 SVC 模式 */
10     mrs r0, cpsr
11     bic r0, r0, #0x1f   /* 将 r0 的低 5 位清零，也就是 cpsr 的 M0~M4    */
12     orr r0, r0, #0x13   /* r0 或上 0x13,表示使用 SVC 模式                */
```

```
13      msr cpsr, r0          /* 将 r0 的数据写入到 cpsr_c 中                */
14
15      ldr sp, =0X80200000   /* 设置栈指针              */
16      b main                /* 跳转到 main 函数        */
```

第 1 行定义了一个全局标号_start。

第 7 行就是标号_start 开始的地方，相当于是一个_start 函数，这个_start 就是第一行代码。

第 10~13 行就是设置处理器进入 SVC 模式，在 6.2 小节的“Cortex-A 处理器运行模型”中我们说过 Cortex-A 有九个运行模型，这里我们设置处理器运行在 SVC 模式下。处理器模式的设置是通过修改 CPSR(程序状态)寄存器来完成的，在 6.3.2 小节中我们详细的讲解了 CPSR 寄存器，其中 M[4:0](CPSR 的 bit[4:0])就是设置处理器运行模式的，参考表 6.3.2.2，如果要将处理器设置为 SVC 模式，那么 M[4:0]就要等于 0X13。11~13 行代码就是先使用指令 MRS 将 CPSR 寄存器的值读取到 R0 中，然后修改 R0 中的值，设置 R0 的 bit[4:0]为 0X13，然后再使用指令 MSR 将修改后的 R0 重新写入到 CPSR 中。

第 15 行通过 ldr 指令设置 SVC 模式下的 SP 指针=0X80200000，因为 I.MX6U-ALPHA 开发板上的 DDR3 地址范围是 0X80000000~0XA0000000(512MB) 或者 0X80000000~0X90000000(256MB)，不管是 512MB 版本还是 256MB 版本的，其 DDR3 起始地址都是 0X80000000。由于 Cortex-A7 的堆栈是向下增长的，所以将 SP 指针设置为 0X80200000，因此 SVC 模式的栈大小 0X80200000-0X80000000=0X200000=2MB，2MB 的栈空间已经很大了，如果做裸机开发的话绰绰有余。

第 15 行就是跳转到 main 函数，main 函数就是 C 语言代码了。

至此汇编部分程序执行完成，就几行代码，用来设置处理器运行到 SVC 模式下、然后初始化 SP 指针、最终跳转到 C 文件的 mian 函数中。如果有玩过三星的 S3C2440 或者 S5PV210 的话会知道我们在使用 SDRAM 或者 DDR 之前必须先初始化 SDRAM 或者 DDR。所以 S3C2440 或者 S5PV210 的汇编文件里面是一定会有 SDRAM 或者 DDR 初始化代码的。我们上面编写的 start.s 文件中却没有初始化 DDR3 的代码，但是却将 SVC 模式下的 SP 指针设置到了 DDR3 的地址范围中，这不会出问题吗？肯定不会的，DDR3 肯定是要初始化的，但是不需要在 start.s 文件中完成。在 9.4.2 小节里面分析 DCD 数据的时候就已经讲过了，DCD 数据包含了 DDR 配置参数，I.MX6U 内部的 Boot ROM 会读取 DCD 数据中的 DDR 配置参数然后完成 DDR 初始化的。

10.3.2 C 语言部分实验程序编写

C 语言部分有两个文件 main.c 和 main.h，main.h 里面主要是定义的寄存器地址，在 mian.h 里面输入代码：

示例代码 10.3.2.1 main.h 文件代码

```
#ifndef __MAIN_H
#define __MAIN_H
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : main.h
作者     : 左忠凯
版本     : V1.0
描述     : 时钟 GPIO1_IO03 相关寄存器地址定义。
其他     : 无
```

```
日志      : 初版 V1.0 2019/1/3 左忠凯创建
***************************************************************/

1  /*
2   * CCM 相关寄存器地址
3   */
4  #define CCM_CCGR0              *((volatile unsigned int *)0X020C4068)
5  #define CCM_CCGR1              *((volatile unsigned int *)0X020C406C)
6  #define CCM_CCGR2              *((volatile unsigned int *)0X020C4070)
7  #define CCM_CCGR3              *((volatile unsigned int *)0X020C4074)
8  #define CCM_CCGR4              *((volatile unsigned int *)0X020C4078)
9  #define CCM_CCGR5              *((volatile unsigned int *)0X020C407C)
10 #define CCM_CCGR6              *((volatile unsigned int *)0X020C4080)
11
12 /*
13  * IOMUX 相关寄存器地址
14  */
15 #define SW_MUX_GPIO1_IO03      *((volatile unsigned int *)0X020E0068)
16 #define SW_PAD_GPIO1_IO03      *((volatile unsigned int *)0X020E02F4)
17
18 /*
19  * GPIO1 相关寄存器地址
20  */
21 #define GPIO1_DR               *((volatile unsigned int *)0X0209C000)
22 #define GPIO1_GDIR             *((volatile unsigned int *)0X0209C004)
23 #define GPIO1_PSR              *((volatile unsigned int *)0X0209C008)
24 #define GPIO1_ICR1             *((volatile unsigned int *)0X0209C00C)
25 #define GPIO1_ICR2             *((volatile unsigned int *)0X0209C010)
26 #define GPIO1_IMR              *((volatile unsigned int *)0X0209C014)
27 #define GPIO1_ISR              *((volatile unsigned int *)0X0209C018)
28 #define GPIO1_EDGE_SEL         *((volatile unsigned int *)0X0209C01C)
29
30 #endif
```

在 mian.h 中我们以宏定义的形式定义了要使用到的所有寄存器，后面的数字就是其地址，比如 CCM_CCGR0 寄存器的地址就是 0X020C4068，这个很简单，很好理解。

接下看一下 main.c 文件，在 mian.c 里面输入如下所示代码：

示例代码 10.3.2.2 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   :  mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 2 C 语言点灯
```

```
            使用 C 语言来点亮开发板上的 LED 灯，学习和掌握如何用 C 语言来
            完成对 I.MX6U 处理器的 GPIO 初始化和控制。
其他      : 无
日志      : 初版 V1.0 2019/1/3 左忠凯创建
**************************************************************/
1   #include "main.h"
2   
3   /*
4    * @description : 使能 I.MX6U 所有外设时钟
5    * @param       : 无
6    * @return      : 无
7    */
8   void clk_enable(void)
9   {
10      CCM_CCGR0 = 0xffffffff;
11      CCM_CCGR1 = 0xffffffff;
12      CCM_CCGR2 = 0xffffffff;
13      CCM_CCGR3 = 0xffffffff;
14      CCM_CCGR4 = 0xffffffff;
15      CCM_CCGR5 = 0xffffffff;
16      CCM_CCGR6 = 0xffffffff;
17  }
18  
19  /*
20   * @description : 初始化 LED 对应的 GPIO
21   * @param       : 无
22   * @return      : 无
23   */
24  void led_init(void)
25  {
26      /* 1、初始化 IO 复用, 复用为 GPIO1_IO03 */
27      SW_MUX_GPIO1_IO03 = 0x5;
28  
29      /* 2、配置 GPIO1_IO03 的 IO 属性
30       *bit 16:0 HYS 关闭
31       *bit [15:14]: 00 默认下拉
32       *bit [13]: 0 kepper 功能
33       *bit [12]: 1 pull/keeper 使能
34       *bit [11]: 0 关闭开路输出
35       *bit [7:6]: 10 速度 100Mhz
36       *bit [5:3]: 110 R0/6 驱动能力
37       *bit [0]: 0 低转换率
38       */
```

```
39      SW_PAD_GPIO1_IO03 = 0X10B0;
40  
41      /* 3、初始化 GPIO, GPIO1_IO03 设置为输出 */
42      GPIO1_GDIR = 0X0000008;
43  
44      /* 4、设置 GPIO1_IO03 输出低电平，打开 LED0 */
45      GPIO1_DR = 0X0;
46  }
47  
48  /*
49   * @description : 打开 LED 灯
50   * @param       : 无
51   * @return      : 无
52   */
53  void led_on(void)
54  {
55      /*
56       * 将 GPIO1_DR 的 bit3 清零
57       */
58      GPIO1_DR &= ~(1<<3);
59  }
60  
61  /*
62   * @description : 关闭 LED 灯
63   * @param       : 无
64   * @return      : 无
65   */
66  void led_off(void)
67  {
68      /*
69       * 将 GPIO1_DR 的 bit3 置 1
70       */
71      GPIO1_DR |= (1<<3);
72  }
73  
74  /*
75   * @description : 短时间延时函数
76   * @param - n   : 要延时循环次数(空操作循环次数，模式延时)
77   * @return      : 无
78   */
79  void delay_short(volatile unsigned int n)
80  {
81      while(n--){}
```

```
82 }
83
84 /*
85  * @description : 延时函数,在 396Mhz 的主频下延时时间大约为 1ms
86  * @param - n   : 要延时的 ms 数
87  * @return      : 无
88  */
89 void delay(volatile unsigned int n)
90 {
91     while(n--)
92     {
93         delay_short(0x7ff);
94     }
95 }
96
97 /*
98  * @description : mian 函数
99  * @param       : 无
100  * @return      : 无
101  */
102 int main(void)
103 {
104     clk_enable();      /* 使能所有的时钟        */
105     led_init();        /* 初始化 led            */
106
107     while(1)           /* 死循环                */
108     {
109         led_off();     /* 关闭 LED              */
110         delay(500);    /* 延时大约 500ms        */
111
112         led_on();      /* 打开 LED              */
113         delay(500);    /* 延时大约 500ms        */
114     }
115
116     return 0;
117 }
```

main.c 文件里面一共有 7 个函数，这 7 个函数都很简单。clk_enable 函数是使能 CCGR0~CCGR6 所控制的所有外设时钟。led_init 函数是初始化 LED 灯所使用的 IO，包括设置 IO 的复用功能、IO 的属性配置和 GPIO 功能，最终控制 GPIO 输出低电平来打开 LED 灯。led_on 和 led_off 这两个函数看名字就知道，用来控制 LED 灯的亮灭的。delay_short()和 delay()这两个函数是延时函数，delay_short()函数是靠空循环来实现延时的，delay()是对 delay_short()的简单封装，在 I.MX6U 工作在 396MHz(Boot ROM 设置的 396MHz)的主频的时候 delay_short(0x7ff)基本能够实现大约 1ms 的延时，所以 delay()函数我们可以用来完成 ms 延时。

main 函数就是我们的主函数了，在 main 函数中先调用函数 clk_enable()和 led_init()来完成时钟使能和 LED 初始化，最终在 while(1)循环中实现 LED 循环亮灭，亮灭时间大约是 500ms。

本实验的程序部分就是这些了，接下来即使编译和测试了。

10.4 编译下载验证

10.4.1 编写 Makefile

新建 Makefile 文件，在 Makefile 文件里面输入如下内容：

示例代码 10.3.2.2 main.c 文件代码

```
1  objs := start.o main.o
2
3  ledc.bin:$(objs)
4   arm-linux-gnueabihf-ld -Ttext 0X87800000 -o ledc.elf $^
5   arm-linux-gnueabihf-objcopy -O binary -S ledc.elf $@
6   arm-linux-gnueabihf-objdump -D -m arm ledc.elf > ledc.dis
7
8  %.o:%.s
9   arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
10
11 %.o:%.S
12  arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
13
14 %.o:%.c
15  arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
16
17 clean:
18  rm -rf *.o ledc.bin ledc.elf ledc.dis
```

上述的 Makefile 就比第八章的 Makefile 要复杂一点了，里面用到了 Makefile 变量和自动变量，关于 Makefile 的变量和自动变量的请参考“3.4 Makefile 语法”。

第 1 行定义了一个变量 objs，objs 包含着要生成 ledc.bin 所需的材料：start.o 和 main.o，也就是当前工程下的 start.s 和 main.c 这两个文件编译后的.o 文件。这里要注意 start.o 一定要放到最前面！因为在后面链接的时候 start.o 要在最前面，因为 start.o 是最先要执行的文件！

第 3 行就是默认目标，目的是生成最终的可执行文件 ledc.bin，ledc.bin 依赖 start.o 和 main.o 如果当前工程没有 start.o 和 main.o 的时候就会找到相应的规则去生成 start.o 和 main.o。比如 start.o 是 start.s 文件编译生成的，因此会执行第 8 行的规则。

第 4 行是使用 arm-linux-gnueabihf-ld 进行链接，链接起始地址是 0X87800000，但是这一行用到了自动变量“$^”，“$^”的意思是所有依赖文件的集合，在这里就是 objs 这个变量的值：start.o 和 main.o。链接的时候 start.o 要链接到最前面，因为第一行代码就是 start.o 里面的，因此这一行就相当于：

```
arm-linux-gnueabihf-ld -Ttext 0X87800000 -o ledc.elf start.o main.o
```

第 5 行使用 arm-linux-gnueabihf-objcopy 来将 ledc.elf 文件转为 ledc.bin，本行也用到了自动变量“$@”，“$@”的意思是目标集合，在这里就是“ledc.bin”，那么本行就相当于：

```
arm-linux-gnueabihf-objcopy -O binary -S ledc.elf ledc.bin
```

第 6 行使用 arm-linux-gnueabihf-objdump 来反汇编，生成 ledc.dis 文件。

第 8~15 行就是针对不同的文件类型将其编译成对应的.o 文件，其实就是汇编.s(.S)和.c 文件，比如 start.s 就会使用第 8 行的规则来生成对应的 start.o 文件。第 9 行就是具体的命令，这行也用到了自动变量“$@”和“$<”，其中“$<”的意思是依赖目标集合的第一个文件。比如 start.s 要编译成 start.o 的话第 8 行和第 9 行就相当于：

```
start.o:start.s
    arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o start.o start.s
```

第 17 行就是工程清理规则，通过命令“make clean”就可以清理工程。

Makefile 文件就讲到这里，我们可以将整个工程拿到 Ubuntu 下去编译，编译完成以后可以使用软件 imxdownload 将其下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                  //给予 imxdownoad 可执行权限，一次即可
./imxdownload ledc.bin /dev/sdd        //下载到 SD 卡中
```

10.4.2 链接脚本

在上面的 Makefile 中我们链接代码的时候使用如下语句：

```
arm-linux-gnueabihf-ld -Ttext 0X87800000 -o ledc.elf $^
```

上面语句中我们是通过“-Ttext”来指定链接地址是 0X87800000 的，这样的话所有的文件都会链接到以 0X87800000 为起始地址的区域。但是有时候我们很多文件需要链接到指定的区域，或者叫做段里面，比如在 Linux 里面初始化函数就会放到 init 段里面。因此我们需要能够自定义一些段，这些段的起始地址我们可以自由指定，同样的我们也可以指定一个文件或者函数应该存放到哪个段里面去。要完成这个功能我们就需要使用到链接脚本，看名字就知道链接脚本主要用于链接的，用于描述文件应该如何被链接在一起形成最终的可执行文件。其主要目的是描述输入文件中的段如何被映射到输出文件中，并且控制输出文件中的内存排布。比如我们编译生成的文件一般都包含 text 段、data 段等等。

链接脚本的语法很简单，就是编写一系列的命令，这些命令组成了链接脚本，每个命令是一个带有参数的关键字或者一个对符号的赋值，可以使用分号分隔命令。像文件名之类的字符串可以直接键入，也可以使用通配符“*”。最简单的链接脚本可以只包含一个命令“SECTIONS”，我们可以在这一个“SECTIONS”里面来描述输出文件的内存布局。我们一般编译出来的代码都包含在 text、data、bss 和 rodata 这四个段内，假设现在的代码要被链接到 0X10000000 这个地址，数据要被链接到 0X30000000 这个地方，下面就是完成此功能的最简单的链接脚本：

示例代码 10.4.2.1 链接脚本演示代码

```
1 SECTIONS{
2   . = 0X10000000;
3   .text : {*(.text)}
4   . = 0X30000000;
5   .data ALIGN(4) : { *(.data) }
6   .bss ALIGN(4)  : { *(.bss) }
7 }
```

第 1 行我们先写了一个关键字“SECTIONS”，后面跟了一个大括号，这个大括号和第 7 行的大括号是一对，这是必须的。看起来就跟 C 语言里面的函数一样。

第 2 行对一个特殊符号“.”进行赋值，“.”在链接脚本里面叫做定位计数器，默认的定位计数器为 0。我们要求代码链接到以 0X10000000 为起始地址的地方，因此这一行给“.”赋值

0X10000000，表示以 0X10000000 开始，后面的文件或者段都会以 0X10000000 为起始地址开始链接。

第 3 行的“.text”是段名，后面的冒号是语法要求，冒号后面的大括号里面可以填上要链接到“.text”这个段里面的所以文件，“*(.text)”中的“*”是通配符，表示所有输入文件的.text 段都放到“.text”中。

第 4 行，我们的要求是数据放到 0X30000000 开始的地方，所以我们需要重新设置定位计数器“.”，将其改为 0X30000000。如果不重新设置的话会怎么样？假设“.text”段大小为 0X10000，那么接下来的.data 段开始地址就是 0X10000000+0X10000=0X10010000，这明显不符合我们的要求。所以我们必须调整定位计数器为 0X30000000。

第 5 行跟第 3 行一样，定义了一个名为“.data”的段，然后所有文件的“.data”段都放到这里面。但是这一行多了一个“ALIGN(4)”，这是什么意思呢？这是用来对“.data”这个段的起始地址做字节对齐的，ALIGN(4)表示 4 字节对齐。也就是说段“.data”的起始地址要能被 4 整除，一般常见的都是 ALIGN(4)或者 ALIGN(8)，也就是 4 字节或者 8 字节对齐。

第 6 行定义了一个“.bss”段，所有文件中的“.bss”数据都会被放到这个里面，“.bss”数据就是那些定义了但是没有被初始化的变量。

上面就是链接脚本最基本的语法格式，我们接下来就按照这个基本的语法格式来编写我们本试验的链接脚本，我们本试验的链接脚本要求如下：

、链接起始地址为 0X87800000。

、start.o 要被链接到最开始的地方，因为 start.o 里面包含这第一个要执行的命令。

根据要求，在 Makefile 同目录下新建一个名为“imx6ul.lds”的文件，然后在此文件里面输入如下所示代码：

示例代码 10.4.2.2 imx6ul.lds 链接脚本代码

```
1  SECTIONS{
2      . = 0X87800000;
3      .text :
4      {
5          start.o
6          main.o
7          *(.text)
8      }
9      .rodata ALIGN(4) : {*(.rodata*)}
10     .data ALIGN(4)   : { *(.data) }
11     __bss_start = .;
12     .bss ALIGN(4)  : { *(.bss)  *(COMMON) }
13     __bss_end = .;
14 }
```

上面的链接脚本文件和示例代码 10.4.2.1 基本一致的，第 2 行设置定位计数器为 0X87800000，因为我们的链接地址就是 0X87800000。第 5 行设置链接到开始位置的文件为 start.o，因为 start.o 里面包含着第一个要执行的指令，所以一定要链接到最开始的地方。第 6 行是 main.o 这个文件，其实可以不用写出来，因为 main.o 的位置就无所谓了，可以由编译器自行决定链接位置。在第 11、13 行有“__bss_start”和“__bss_end”这两个东西？这个是什么呢？“__bss_start”和“__bss_end”是符号，第 11、13 这两行其实就是对这两个符号进行赋值，其值为定位符“.”，这两个符号用来保存.bss 段的起始地址和结束地址。前面说了.bss 段是定义了但是没有被初始

化的变量，我们需要手动对.bss 段的变量清零的，因此我们需要知道.bss 段的起始和结束地址，这样我们直接对这段内存赋 0 即可完成清零。通过第 11、13 行代码，.bss 段的起始地址和结束地址就保存在了“__bss_start”和“__bss_end”中，我们就可以直接在汇编或者 C 文件里面使用这两个符号。

10.4.3 修改 Makefile

在上一小节中我们已经编写好了链接脚本文件：imx6ul.lds，我们肯定是要使用这个链接脚本文件的，将 Makefile 中的如下一行代码：

```
arm-linux-gnueabihf-ld -Ttext 0X87800000 -o ledc.elf $^
```

改为：

```
arm-linux-gnueabihf-ld -Timx6ul.lds -o ledc.elf $^
```

起始就是将-T 后面的 0X87800000 改为 imx6ul.lds，表示使用 imx6ul.lds 这个链接脚本文件。修改完成以后使用新的 Makefile 和链接脚本文件重新编译工程，编译成功以后就可以烧写到 SD 卡中验证了。

10.4.4 下载验证

使用软件 imxdownload 将编译出来的 ledc.bin 烧写到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload ledc.bin /dev/sdd          //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板，如果代码运行正常的话 LED0 就会以 500ms 的时间间隔亮灭。

第十一章 模仿 STM32 驱动开发格式实验

在上一章使用 C 语言编写 LED 灯驱动的时候，每个寄存器的地址我们都需要写宏定义，使用起来非常的不方便。我们在学习 STM32 的时候，可以使用“GPIOB->ODR”这种方式来给 GPIOB 的寄存器 ODR 赋值，因为在 STM32 中同属于一个外设的所有寄存器地址基本是相邻的(有些会有保留寄存器)。因此我们可以借助 C 语言里面的结构体成员地址递增的特点来将某个外设的所有寄存器写入到一个结构体里面，然后定义一个结构体指针指向这个外设的寄存器基地址，这样我们就可以通过这个结构体指针来访问这个外设的所有寄存器。同理，I.MX6U 也可以使用这种方法来定义外设寄存器，本章我们就模仿 STM32 里面的寄存器定义方式来编写 I.MX6U 的驱动，通过本章的学习也可以对 STM32 的寄存器定义方式有一个深入的认识。

11.1 模仿 STM32 寄存器定义

11.1.1 STM32 寄存器定义简介

为了开发方便，ST 官方为 STM32F103 编写了一个叫做 stm32f10x.h 的文件，在这个文件里面定义了 STM32F103 所有外设寄存器，我们可以使用其定义的寄存器来进行开发，比如我们可以用如下代码来初始化一个 GPIO：

```
GPIOE->CRL&=0XFF0FFFFF;
GPIOE->CRL|=0X00300000;          //PE5 推挽输出
GPIOE->ODR|=1<<5;                //PE5 输出高
```

上述代码是初始化 STM32 的 PE5 这个 GPIO 为推挽输出，需要配置的就是 GPIOE 的寄存器 CRL 和 ODR，“GPIOE”的定义：

```
#define GPIOE                  ((GPIO_TypeDef *) GPIOE_BASE)
```

可以看出“GPIOE”是个宏定义，是一个指向地址 GPIOE_BASE 的结构体指针，结构体为 GPIO_TypeDef，GPIO_TypeDef 和 GPIOE_BASE 的定义如下：

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;

#define GPIOE_BASE             (APB2PERIPH_BASE + 0x1800)
#define APB2PERIPH_BASE        (PERIPH_BASE + 0x10000)
#define PERIPH_BASE            ((uint32_t)0x40000000)
```

上述定义中 GPIO_TypeDef 是个结构体，结构体里面的成员变量有 CRL、CRH、IDR、ODR、BSRR、BRR 和 LCKR，这些都是 GPIO 的寄存器，每个成员变量都是 32 位(4 字节)，这些寄存器在结构体中的位置都是按照其地址值从小到大排序的。GPIOE_BASE 就是 GPIOE 的基地址，其为：

```
GPIOE_BASE=APB2PERIPH_BASE+0x1800
            = PERIPH_BASE + 0x10000 + 0x1800
            =0x40000000 + 0x10000 + 0x1800
            =0x40011800
```

GPIOE_BASE 的基地址为 0x40011800，宏 GPIOE 指向这个地址，因此 GPIOE 的寄存器 CRL 的地址就是 0X40011800，寄存器 CRH 的地址就是 0X40011800+4=0X40011804，其他寄存器地址以此类推。我们要操作 GPIOE 的 ODR 寄存器的话就可以通过“GPIOE->ODR”来实现，这个方法是借助了结构体成员地址连续递增的原理。

了解了 STM32 的寄存器定义以后，我们就可以参考其原理来编写 I.MX6U 的外设寄存器定义了。NXP 官方并没有为 I.MX6UL 编写类似 stm32f10x.h 这样的文件，NXP 只为 I.MX6ULL

提供了类似 stm32f10x.h 这样的文件，名为 MCIMX6Y2.h，但是 I.MX6UL 和 I.MX6ULL 几乎一模一样，所以文件 MCIMX6Y2.h 可以用在 I.MX6UL 上。关于文件 MCIMX6Y2.h 的移植我们在下一章讲解，本章我们参考 stm32f10x.h 来编写一个简单的 MCIMX6Y2.h 文件。

11.1.2 I.MX6U 寄存器定义

参考 STM32 的官方文件来编写 I.MX6U 的寄存器定义，比如 IO 复用寄存器组“IOMUX_SW_MUX_CTL_PAD_XX”，步骤如下：

1、编写外设结构体

先将同属于一个外设的所有寄存器编写到一个结构体里面，如 IO 复用寄存器组的结构体如下：

示例代码 11.1.2.1 寄存器 IOMUX_SW_MUX_Type

```
/*
 * IOMUX 寄存器组
 */
1   typedef struct
2   {
3       volatile unsigned int BOOT_MODE0;
4       volatile unsigned int BOOT_MODE1;
5       volatile unsigned int SNVS_TAMPER0;
6       volatile unsigned int SNVS_TAMPER1;
        .........
107     volatile unsigned int CSI_DATA00;
108     volatile unsigned int CSI_DATA01;
109     volatile unsigned int CSI_DATA02;
110     volatile unsigned int CSI_DATA03;
111     volatile unsigned int CSI_DATA04;
112     volatile unsigned int CSI_DATA05;
113     volatile unsigned int CSI_DATA06;
114     volatile unsigned int CSI_DATA07;
        /* 为了缩短代码，其余 IO 复用寄存器省略 */
115}IOMUX_SW_MUX_Tpye;
```

上述结构体 IOMUX_SW_MUX_Type 就是 IO 复用寄存器组，成员变量是每个 IO 对应的复用寄存器，每个寄存器的地址是 32 位，每个成员都使用“volatile”进行了修饰，目的是防止编译器优化。

2、定义 IO 复用寄存器组的基地址

根据结构体 IOMUX_SW_MUX_Type 的定义，其第一个成员变量为 BOOT_MODE0，也就是 BOOT_MODE0 这个 IO 的 IO 复用寄存器，查找 I.MX6U 的参考手册可以得知其地址为 0X020E0014，所以 IO 复用寄存器组的基地址就是 0X020E0014，定义如下：

```
#define IOMUX_SW_MUX_BASE            (0X020E0014)
```

3、定义访问指针

访问指针定义如下：

```
#define IOMUX_SW_MUX        ((IOMUX_SW_MUX_Type *)IOMUX_SW_MUX_BASE)
```

通过上面三步我们就可以通过“IOMUX_SW_MUX->GPIO1_IO03”来访问 GPIO1_IO03 的 IO 复用寄存器了。同样的，其他的外设寄存器都可以通过这三步来定义。

11.2 硬件原理分析

本章使用到的硬件资源和第八章一样，就是一个 LED0。

11.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 3_ledc_stm32。

创建 VSCode 工程，工作区名字为“ledc_stm32”，新建三个文件：start.S、main.c 和 imx6ul.h。其中 start.S 是汇编文件，start.S 文件的内容和第十章的 start.S 一样，直接复制过来就可以。main.c 和 imx6ul.h 是 C 文件，完成以后如图 11.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/3_ledc_stm32$ ls -a
.  ..  imx6ul.h  main.c  start.S  .vscode
zuozhongkai@ubuntu:~/linux/3_ledc_stm32$
```

图 11.3.1 工程文件目录

文件 imx6ul.h 用来存放外设寄存器定义，在 imx6ul.h 中输入如下代码：

示例代码 11.2.1 imx6ul.h 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : imx6ul.h
作者      : 左忠凯
版本      : V1.0
描述      : IMX6UL 相关寄存器定义，参考 STM32 寄存器定义方法
其他      : 无
日志      : 初版 V1.0 2019/1/3 左忠凯创建
**************************************************************/


/*
 * 外设寄存器组的基地址
 */
1   #define CCM_BASE                (0X020C4000)
2   #define CCM_ANALOG_BASE         (0X020C8000)
3   #define IOMUX_SW_MUX_BASE       (0X020E0014)
4   #define IOMUX_SW_PAD_BASE       (0X020E0204)
5   #define GPIO1_BASE              (0x0209C000)
6   #define GPIO2_BASE              (0x020A0000)
7   #define GPIO3_BASE              (0x020A4000)
8   #define GPIO4_BASE              (0x020A8000)
9   #define GPIO5_BASE              (0x020AC000)
10
11   /*
12    * CCM 寄存器结构体定义，分为 CCM 和 CCM_ANALOG
13    */
```

```
14  typedef struct
15  {
16      volatile unsigned int CCR;
17      volatile unsigned int CCDR;
18      volatile unsigned int CSR;
        ……
46      volatile unsigned int CCGR6;
47      volatile unsigned int RESERVED_3[1];
48      volatile unsigned int CMEOR;
49  } CCM_Type;
50
51  typedef struct
52  {
53      volatile unsigned int PLL_ARM;
54      volatile unsigned int PLL_ARM_SET;
55      volatile unsigned int PLL_ARM_CLR;
56      volatile unsigned int PLL_ARM_TOG;
        ……
110     volatile unsigned int MISC2;
111     volatile unsigned int MISC2_SET;
112     volatile unsigned int MISC2_CLR;
113     volatile unsigned int MISC2_TOG;
114 } CCM_ANALOG_Type;
115
116 /*
117  * IOMUX 寄存器组
118  */
119 typedef struct
120 {
121     volatile unsigned int BOOT_MODE0;
122     volatile unsigned int BOOT_MODE1;
123     volatile unsigned int SNVS_TAMPER0;
        ……
241     volatile unsigned int CSI_DATA04;
242     volatile unsigned int CSI_DATA05;
243     volatile unsigned int CSI_DATA06;
244     volatile unsigned int CSI_DATA07;
245 }IOMUX_SW_MUX_Type;
246
247 typedef struct
248 {
249     volatile unsigned int DRAM_ADDR00;
250     volatile unsigned int DRAM_ADDR01;
```

```
        ......
419     volatile unsigned int GRP_DDRPKE;
420     volatile unsigned int GRP_DDRMODE;
421     volatile unsigned int GRP_DDR_TYPE;
422 }IOMUX_SW_PAD_Type;
423 
424 /* 
425  * GPIO 寄存器结构体
426  */
427 typedef struct
428 {
429     volatile unsigned int DR;
430     volatile unsigned int GDIR;
431     volatile unsigned int PSR;
432     volatile unsigned int ICR1;
433     volatile unsigned int ICR2;
434     volatile unsigned int IMR;
435     volatile unsigned int ISR;
436     volatile unsigned int EDGE_SEL;
437 }GPIO_Type;
438 
439 
440 /* 
441  * 外设指针
442  */
443 #define CCM             ((CCM_Type *)CCM_BASE)
444 #define CCM_ANALOG      ((CCM_ANALOG_Type *)CCM_ANALOG_BASE)
445 #define IOMUX_SW_MUX    ((IOMUX_SW_MUX_Type *)IOMUX_SW_MUX_BASE)
446 #define IOMUX_SW_PAD    ((IOMUX_SW_PAD_Type *)IOMUX_SW_PAD_BASE)
447 #define GPIO1           ((GPIO_Type *)GPIO1_BASE)
448 #define GPIO2           ((GPIO_Type *)GPIO2_BASE)
449 #define GPIO3           ((GPIO_Type *)GPIO3_BASE)
450 #define GPIO4           ((GPIO_Type *)GPIO4_BASE)
451 #define GPIO5           ((GPIO_Type *)GPIO5_BASE)
```

在编写寄存器组结构体的时候注意寄存器的地址是否连续，有些外设的寄存器地址可能不是连续的，会有一些保留地址，因此我们需要在结构体中留出这些保留的寄存器。比如 CCM 的 CCGR6 寄存器地址为 0X020C4080，而寄存器 CMEOR 的地址为 0X020C4088。按照地址顺序递增的原理，寄存器 CMEOR 的地址应该是 0X020C4084，但是实际上 CMEOR 的地址是 0X020C4088，相当于中间跳过了 0X020C4088-0X020C4080=8 个字节，如果寄存器地址连续的话应该只差 4 个字节(32 位)，但是现在差了 8 个字节，所以需要在寄存器 CCGR6 和 CMEOR 直接加入一个保留寄存器，这个就是“示例代码 11.3.1”中第 47 行 RESERVED_3[1]的来源。如果不添加保留为来占位的话就会导致寄存器地址错位！

main.c 文件中输入如下所示内容：

示例代码 11.3.2　main.c 文件代码

```
1   #include "imx6ul.h"
2   
3   /*
4    * @description : 使能 I.MX6U 所有外设时钟
5    * @param       : 无
6    * @return      : 无
7    */
8   void clk_enable(void)
9   {
10      CCM->CCGR0 = 0XFFFFFFFF;
11      CCM->CCGR1 = 0XFFFFFFFF;
12      CCM->CCGR2 = 0XFFFFFFFF;
13      CCM->CCGR3 = 0XFFFFFFFF;
14      CCM->CCGR4 = 0XFFFFFFFF;
15      CCM->CCGR5 = 0XFFFFFFFF;
16      CCM->CCGR6 = 0XFFFFFFFF;
17  }
18  
19  /*
20   * @description : 初始化 LED 对应的 GPIO
21   * @param       : 无
22   * @return      : 无
23   */
24  void led_init(void)
25  {
26      /* 1、初始化 IO 复用 */
27      IOMUX_SW_MUX->GPIO1_IO03 = 0X5;     /* 复用为 GPIO1_IO03 */
28  
29  
30      /* 2、配置 GPIO1_IO03 的 IO 属性
31       *bit 16:0 HYS 关闭
32       *bit [15:14]: 00 默认下拉
33       *bit [13]: 0 kepper 功能
34       *bit [12]: 1 pull/keeper 使能
35       *bit [11]: 0 关闭开路输出
36       *bit [7:6]: 10 速度 100Mhz
37       *bit [5:3]: 110 R0/6 驱动能力
38       *bit [0]: 0 低转换率
39       */
40      IOMUX_SW_PAD->GPIO1_IO03 = 0X10B0;
41  
42  
```

```
43      /* 3、初始化 GPIO */
44      GPIO1->GDIR = 0X0000008;    /* GPIO1_IO03 设置为输出 */
45
46      /* 4、设置 GPIO1_IO03 输出低电平，打开 LED0 */
47      GPIO1->DR &= ~(1 << 3);
48
49  }
50
51  /*
52   * @description : 打开 LED 灯
53   * @param       : 无
54   * @return      : 无
55   */
56  void led_on(void)
57  {
58      /* 将 GPIO1_DR 的 bit3 清零     */
59      GPIO1->DR &= ~(1<<3);
60  }
61
62  /*
63   * @description : 关闭 LED 灯
64   * @param       : 无
65   * @return      : 无
66   */
67  void led_off(void)
68  {
69      /* 将 GPIO1_DR 的 bit3 置 1 */
70      GPIO1->DR |= (1<<3);
71  }
72
73  /*
74   * @description : 短时间延时函数
75   * @param - n    : 要延时循环次数(空操作循环次数，模式延时)
76   * @return      : 无
77   */
78  void delay_short(volatile unsigned int n)
79  {
80      while(n--){}
81  }
82
83  /*
84   * @description : 延时函数,在 396Mhz 的主频下
85   *                延时时间大约为 1ms
```

```
86   * @param - n   : 要延时的 ms 数
87   * @return      : 无
88   */
89  void delay(volatile unsigned int n)
90  {
91      while(n--)
92      {
93          delay_short(0x7ff);
94      }
95  }
96
97  /*
98   * @description : mian 函数
99   * @param       : 无
100  * @return      : 无
101  */
102 int main(void)
103 {
104     clk_enable();           /* 使能所有的时钟       */
105     led_init();             /* 初始化 led           */
106
107     while(1)                /* 死循环               */
108     {
109         led_off();          /* 关闭 LED             */
110         delay(500);         /* 延时 500ms           */
111
112         led_on();           /* 打开 LED             */
113         delay(500);         /* 延时 500ms           */
114     }
115
116     return 0;
117 }
```

main.c 中 7 个函数，这 7 个函数的含义和第十章中的 main.c 文件一样，只是函数体写法变了，寄存器的访问采用 imx6ul.h 中定义的外设指针。比如第 27 行设置 GPIO1_IO03 的复用功能就可以通过“IOMUX_SW_MUX->GPIO1_IO03”来给寄存 SW_MUX_CTL_PAD_GPIO1_IO03 赋值。

11.4 编译下载验证

11.4.1 编写 Makefile 和链接脚本

Makefile 文件的内容基本和第十章的 Makefile 一样，如下：

示例代码 11.4.1 Makefile 文件代码

```
1  objs := start.o main.o
2
3  ledc.bin:$(objs)
4      arm-linux-gnueabihf-ld -Timx6ul.lds -o ledc.elf $^
5      arm-linux-gnueabihf-objcopy -O binary -S ledc.elf $@
6      arm-linux-gnueabihf-objdump -D -m arm ledc.elf > ledc.dis
7
8  %.o:%.s
9      arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
10
11 %.o:%.S
12     arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
13
14 %.o:%.c
15     arm-linux-gnueabihf-gcc -Wall -nostdlib -c -O2 -o $@ $<
16
17 clean:
18     rm -rf *.o ledc.bin ledc.elf ledc.dis
```

链接脚本 imx6ul.lds 的内容和上一章一样，可以直接使用上一章的链接脚本文件。

11.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 ledc.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload ledc.bin /dev/sdd      //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板，如果代码运行正常的话 LED0 就会以 500ms 的时间间隔亮灭，实验现象和上一章一样。

第十二章 官方 SDK 移植试验

在上一章中，我们参考 ST 官方给 STM32 编写的 stm32f10x.h 来自行编写 I.MX6U 的寄存器定义文件。自己编写这些寄存器定义不仅费时费力，没有任何意义，而且很容易写错，幸好 NXP 官方为 I.MX6ULL 编写了 SDK 包，在 SDK 包里面 NXP 已经编写好了寄存器定义文件，所以我们可以直接移植 SDK 包里面的文件来用。虽然 NXP 是为 I.MX6ULL 编写的 SDK 包，但是 I.MX6UL 也是可以使用的！本章我们就来讲解如何移植 SDK 包里面重要的文件，方便我们的开发。

12.1 I.MX6ULL 官方 SDK 包简介

NXP 针对 I.MX6ULL 编写了一个 SDK 包，这个 SDK 包就类似于 STM32 的 STD 库或者 HAL 库，这个 SDK 包提供了 Windows 和 Linux 两种版本，分别针对主机系统是 Windows 和 Linux。因为我们是在 Windows 下使用 Source Insight 来编写代码的，因此我们使用的是 Windows 版本的。Windows 版本 SDK 里面的例程提供了 IAR 版本，肯定有人会问既然 NXP 提供了 IAR 版本的 SDK，那我们为什么不用 IAR 来完成裸机试验，偏偏要用复杂的 GCC？因为我们要从简单的裸机开始掌握 Linux 下的 GCC 开发方法，包括 Ubuntu 操作系统的使用、Makefile 的编写、shell 等等。如果为了偷懒而使用 IAR 开发裸机的话，那么后续学习 Uboot 移植、Linux 移植和 Linux 驱动开发就会很难上手，因为开发环境都不熟悉！再者，不是所有的半导体厂商都会为 Cortex-A 架构的芯片编写裸机 SDK 包，我使用过那么多的 Cotex-A 系列芯片，也就发现了 NXP 给 I.MX6ULL 编写了裸机 SDK 包。而且去 NXP 官网看一下，会发现只有 I.MX6ULL 这一款 Cotex-A 内核的芯片有裸机 SDK 包，NXP 的其它 Cotex-A 芯片都没有。说明在 NXP 的定位里面，I.MX6ULL 就是一个 Cotex-A 内核的高端单片机，定位类似 ST 的 STM32H7。说这么多的目的就是想告诉大家，使用 Cortex-A 内核芯片的时候不要想着有类似 STM32 库一样的东西，I.MX6ULL 是一个特例，基本所有的 Cortex-A 内核的芯片都不会提供裸机 SDK 包。因此在使用 STM32 的时候那些用起来很顺手的库文件，在 Cotex-A 芯片下基本都需要我们自行编写，比如.s 启动文件、寄存器定义等等。

因为本教程是教大家 Linux 驱动开发入门的，本教程需要尽可能的降低入门难度，这也是为什么本教程会选择 I.MX6U 芯片的一个重要的原因，因为其提供了 I.MX6ULL 的裸机 SDK 包，大家上手会很容易。I.MX6ULL 的 SDK 包在 NXP 官网下载，下载界面如图 12.1.1 所示：



图 12.1.1 I.MX6ULL SDK 包下载界面

我们下载图 12.1.1 中的 WIN 版本 SDK，也就是"SDK2.2_iMX6ULL_WIN"，我们已经下载好放到光盘中，路径为：开发板光盘-> 7、I.MX6U 参考资料->3、I.MX6ULL SDK 包->SDK_2.2_MCIM6ULL_RFP_Win.exe。双击 SDK_2.2_MCIM6ULL_RFP_Win.exe 安装 SDK 包，安装的时候需要设置好安装位置，安装完成以后的 SDK 包如图 12.1.2 所示：

名称	修改日期	类型	大小
boards	2019-02-14 22:43	文件夹	
CMSIS	2019-02-14 22:44	文件夹	
CORTEXA	2019-02-14 22:44	文件夹	
devices	2019-02-14 22:44	文件夹	
docs	2019-02-14 22:44	文件夹	
middleware	2019-02-14 22:44	文件夹	
rtos	2019-02-14 22:44	文件夹	
tools	2019-02-14 22:44	文件夹	
EVK-MCIMX6ULL_manifest.xml	2017-06-07 13:23	XML 文档	459 KB
LA_OPT_Base_License.htm	2017-06-07 13:23	360 se HTML Do...	148 KB
SW-Content-Register.txt	2017-06-07 13:23	文本文档	5 KB

图 12.1.2 SDK 包

我们本教程不是讲解 SDK 包如何开发的，我们只是需要 SDK 包里面的几个文件，所以就不去详细的讲解这个 SDK 包了，感兴趣的可以看一下，所有的例程都在 boards 这个文件夹里面。我们重点是需要 SDK 包里面与寄存器定义相关的文件，一共需要如下三个文件：

fsl_common.h：位置为 SDK_2.2_MCIM6ULL\devices\MCIMX6Y2\drivers\fsl_common.h。

fsl_iomuxc.h： 位置为 SDK_2.2_MCIM6ULL\devices\MCIMX6Y2\drivers\fsl_iomuxc.h。

MCIMX6Y2.h: 位置为 SDK_2.2_MCIM6ULL\devices\MCIMX6Y2\MCIMX6YH2.h。

整个 SDK 包我们就需要上面这三个文件，把这三个文件准备好，我们后面移植要用。

12.2 硬件原理图分析

本章使用到的硬件资源和第八章一样，就是一个 LED0。

12.3 试验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 4_ledc_sdk。

12.3.1 SDK 文件移植

使用 VSCode 新建工程，将 fsl_common.h、fsl_iomuxc.h 和 MCIMX6Y2.h 这三个文件拷贝到工程中，这三个文件直接编译的话肯定会出错的！需要对其做删减，因为这三个文件里面的代码都比较大，所以就不详细列出这三个文件删减以后的内容了。大家可以参考我们提供的裸机例程来修改这三个文件，很简单的。修改完成以后的工程目录如图 12.3.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$ ls -a
.  ..  fsl_common.h  fsl_iomuxc.h  MCIMX6Y2.h  .vscode
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$
```

图 12.3.1.1 工程目录

12.3.2 创建 cc.h 文件

新建一个名为 cc.h 的头文件，cc.h 里面存放一些 SDK 库文件需要使用到的数据类型，在 cc.h 里面输入如下代码：

示例代码 12.3.2.1 cc.h 文件代码

```
1  #ifndef __CC_H
2  #define __CC_H
```

```
3 /***************************************************************
4 Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5 文件名   :   cc.h
6 作者     : 左忠凯
7 版本     : V1.0
8 描述     : 有关变量类型的定义，NXP 官方 SDK 的一些移植文件会用到。
9 其他     : 无
10 日志    : 初版 V1.0 2019/1/3 左忠凯创建
11 ***************************************************************/
12
13 /*
14  * 自定义一些数据类型供库文件使用
15  */
16 #define     __I     volatile
17 #define     __O     volatile
18 #define     __IO    volatile
19
20 #define      ON      1
21 #define      OFF     0
22
23 typedef   signed  char            int8_t;
24 typedef   signed  short  int      int16_t;
25 typedef   signed         int      int32_t;
26 typedef unsigned         char     uint8_t;
27 typedef unsigned  short  int      uint16_t;
28 typedef unsigned         int      uint32_t;
29 typedef unsigned  long   long     uint64_t;
30 typedef   signed  char            s8;
31 typedef   signed  short  int      s16;
32 typedef   signed  int             s32;
33 typedef   signed  long   long     int     s64;
34 typedef unsigned  char            u8;
35 typedef unsigned  short  int      u16;
36 typedef unsigned  int             u32;
37 typedef unsigned  long   long     int     u64;
38
39 #endif
```

在 cc.h 文件只我们定义了很多的数据类型，因为有些第三方库会用到这些变量类型。

12.3.3 编写实验代码

新建 start.S 和 main.c 这两个文件，start.S 文件的内容和上一章一样，直接复制过来就可以，创建完成以后工程目录如图 12.3.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$ ls -a
.  ..  cc.h  fsl_common.h  fsl_iomuxc.h  main.c  MCIMX6Y2.h  start.S  .vscode
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$
```

图 12.3.3.1 工程目录文件

在 main.c 中输入如下所示代码：

示例代码 12.3.3.1 main.c 文件代码

```
/**************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : mian.c
作者     : 左忠凯
版本     : V1.0
描述     : I.MX6U 开发板裸机实验 4 使用 NXP 提供的 I.MX6ULL 官方 IAR SDK 包开发
其他     : 前面其他所有实验中，寄存器定义都是我们自己手写的，但是 I.MX6U
          的寄存器有很多，全部自己写太费时间，而且没意义。NXP 官方提供了
          针对 I.MX6ULL 的 SDK 开发包，是基于 IAR 环境的，这个 SDK 包里面已经提
          供了 I.MX6ULL 所有相关寄存器定义，虽然是针对 I.MX6ULL 编写的，但是同样
          适用于 I.MX6UL。本节我们就将相关的寄存器定义文件移植到 Linux 环境下，
          要移植的文件有：
          fsl_common.h
          fsl_iomuxc.h
          MCIMX6Y2.h
          自定义文件 cc.h
日志     : 初版 V1.0 2019/1/3 左忠凯创建
**************************************************************/
1   #include "fsl_common.h"
2   #include "fsl_iomuxc.h"
3   #include "MCIMX6Y2.h"
4
5   /*
6    * @description : 使能 I.MX6U 所有外设时钟
7    * @param       : 无
8    * @return      : 无
9    */
10  void clk_enable(void)
11  {
12      CCM->CCGR0 = 0XFFFFFFFF;
13      CCM->CCGR1 = 0XFFFFFFFF;
14
15      CCM->CCGR2 = 0XFFFFFFFF;
16      CCM->CCGR3 = 0XFFFFFFFF;
17      CCM->CCGR4 = 0XFFFFFFFF;
18      CCM->CCGR5 = 0XFFFFFFFF;
19      CCM->CCGR6 = 0XFFFFFFFF;
20
```

```
21 }
22 
23 /*
24  * @description : 初始化 LED 对应的 GPIO
25  * @param       : 无
26  * @return      : 无
27  */
28 void led_init(void)
29 {
30     /* 1、初始化 IO 复用 */
31     IOMUXC_SetPinMux(IOMUXC_GPIO1_IO03_GPIO1_IO03,0);
32 
33     /* 2、、配置 GPIO1_IO03 的 IO 属性
34      *bit 16:0 HYS 关闭
35      *bit [15:14]: 00 默认下拉
36      *bit [13]: 0 kepper 功能
37      *bit [12]: 1 pull/keeper 使能
38      *bit [11]: 0 关闭开路输出
39      *bit [7:6]: 10 速度 100Mhz
40      *bit [5:3]: 110 R0/6 驱动能力
41      *bit [0]: 0 低转换率
42      */
43     IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO03_GPIO1_IO03,0X10B0);
44 
45     /* 3、初始化 GPIO,设置 GPIO1_IO03 设置为输出  */
46     GPIO1->GDIR |= (1 << 3);
47 
48     /* 4、设置 GPIO1_IO03 输出低电平，打开 LED0 */
49     GPIO1->DR &= ~(1 << 3);
50 }
51 
52 /*
53  * @description : 打开 LED 灯
54  * @param       : 无
55  * @return      : 无
56  */
57 void led_on(void)
58 {
59     /* 将 GPIO1_DR 的 bit3 清零    */
60     GPIO1->DR &= ~(1<<3);
61 }
62 
63 /*
```

```
64   * @description : 关闭 LED 灯
65   * @param       : 无
66   * @return      : 无
67   */
68  void led_off(void)
69  {
70      /* 将 GPIO1_DR 的 bit3 置 1 */
71      GPIO1->DR |= (1<<3);
72  }
73
74  /*
75   * @description : 短时间延时函数
76   * @param - n   : 要延时循环次数(空操作循环次数，模式延时)
77   * @return      : 无
78   */
79  void delay_short(volatile unsigned int n)
80  {
81      while(n--){}
82  }
83
84  /*
85   * @description : 延时函数,在 396Mhz 的主频下
86   *                延时时间大约为 1ms
87   * @param - n   : 要延时的 ms 数
88   * @return       : 无
89   */
90  void delay(volatile unsigned int n)
91  {
92      while(n--)
93      {
94          delay_short(0x7ff);
95      }
96  }
97
98  /*
99   * @description : mian 函数
100  * @param       : 无
101  * @return      : 无
102  */
103 int main(void)
104 {
105     clk_enable();     /* 使能所有的时钟   */
106     led_init();       /* 初始化 led       */
```

```
107
108     while(1)            /* 死循环          */
109     {
110         led_off();      /* 关闭 LED        */
111         delay(500);     /* 延时 500ms      */
112
113         led_on();       /* 打开 LED        */
114         delay(500);     /* 延时 500ms      */
115     }
116
117     return 0;
118 }
```

和上一章一样，main.c 有 7 个函数，这 7 个函数的含义都一样，只是本例程我们使用的是移植好的 NXP 官方 SDK 里面的寄存器定义。main.c 文件的这 7 个函数的内容都很简单，前面都讲过很多次了，我们重点来看一下 led_init 函数中的第 31 行和第 43 行，这两行的内容如下：

```
IOMUXC_SetPinMux(IOMUXC_GPIO1_IO03_GPIO1_IO03,      0);
IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO03_GPIO1_IO03,   0X10B0);
```

这里使用了两个函数 IOMUXC_SetPinMux 和 IOMUXC_SetPinConfig，其中函数 IOMUXC_SetPinMux 是用来设置 IO 复用功能的，最终肯定设置的是寄存器“IOMUXC_SW_MUX_CTL_PAD_XX”。函数 IOMUXC_SetPinConfig 设置的是 IO 的上下拉、速度等的，也就是寄存器“IOMUXC_SW_PAD_CTL_PAD_XX”，所以上面两个函数其实就是上一章中的：

```
IOMUX_SW_MUX->GPIO1_IO03 = 0X5;
IOMUX_SW_PAD->GPIO1_IO03 = 0X10B0;
```

函数 IOMUXC_SetPinMux 在文件 fsl_iomuxc.h 中定义，函数源码如下：

```
static inline void IOMUXC_SetPinMux(uint32_t muxRegister,
                                    uint32_t muxMode,
                                    uint32_t inputRegister,
                                    uint32_t inputDaisy,
                                    uint32_t configRegister,
                                    uint32_t inputOnfield)
{
    *((volatile uint32_t *)muxRegister) =
        IOMUXC_SW_MUX_CTL_PAD_MUX_MODE(muxMode) |
                IOMUXC_SW_MUX_CTL_PAD_SION(inputOnfield);

    if (inputRegister)
    {
        *((volatile uint32_t *)inputRegister) =
                IOMUXC_SELECT_INPUT_DAISY(inputDaisy);
    }
}
```

函数 IOMUXC_SetPinMux 有 6 个参数，这 6 个参数的函数如下：

muxRegister：IO 的复用寄存器地址，比如 GPIO1_IO03 的 IO 复用寄存器 SW_MUX_CTL_PAD_GPIO1_IO03 的地址为 0X020E0068。

muxMode：IO 复用值，也就是 ALT0~ALT8，对应数字 0~8，比如要将 GPIO1_IO03 设置为 GPIO 功能的话此参数就要设置为 5。

inputRegister：外设输入 IO 选择寄存器地址，有些 IO 在设置为其他的复用功能以后还需要设置 IO 输入寄存器，比如 GPIO1_IO03 要复用为 UART1_RX 的话还需要设置寄存器 UART1_RX_DATA_SELECT_INPUT，此寄存器地址为 0X020E0624。

inputDaisy：寄存器 inputRegister 的值，比如 GPIO1_IO03 要作为 UART1_RX 引脚的话此参数就是 1。

configRegister：未使用，函数 IOMUXC_SetPinConfig 会使用这个寄存器。

inputOnfield：IO 软件输入使能，以 GPIO1_IO03 为例就是寄存器 SW_MUX_CTL_PAD_GPIO1_IO03 的 SION 位(bit4)。如果需要使能 GPIO1_IO03 的软件输入功能的话此参数应该为 1，否则的话就为 0。

IOMUXC_SetPinMux 的函数体很简单，就是根据参数对寄存器 muxRegister 和 inputRegister 进行赋值。在“示例代码 12.3.3.1”中的 31 行使用此函数将 GPIO1_IO03 的复用功能设置为 GPIO，如下：

```
IOMUXC_SetPinMux(IOMUXC_GPIO1_IO03_GPIO1_IO03, 0);
```

第一次看到上面代码的时候肯定会奇怪，为何只有两个参数？不是应该 6 个参数的吗？不要着急，先看一个 IOMUXC_GPIO1_IO03_GPIO1_IO03 是个什么玩意。这是个宏，在文件 fsl_iomuxc.h 中有定义，NXP 的 SDK 库将一个 IO 的所有复用功能都定义了一个宏，比如 GPIO1_IO03 就有如下 9 个宏定义：

```
IOMUXC_GPIO1_IO03_I2C1_SDA
IOMUXC_GPIO1_IO03_GPT1_COMPARE3
IOMUXC_GPIO1_IO03_USB_OTG2_OC
IOMUXC_GPIO1_IO03_USDHC1_CD_B
IOMUXC_GPIO1_IO03_GPIO1_IO03
IOMUXC_GPIO1_IO03_CCM_DI0_EXT_CLK
IOMUXC_GPIO1_IO03_SRC_TESTER_ACK
IOMUXC_GPIO1_IO03_UART1_RX
IOMUXC_GPIO1_IO03_UART1_TX
```

上面 9 个宏定义分别对应着 GPIO1_IO03 的九种复用功能，比如复用为 GPIO 的宏定义就是：

```
#define IOMUXC_GPIO1_IO03_GPIO1_IO03   0x020E0068U,     0x5U,     0x00000000U,
                                       0x0U,            0x020E02F4U
```

将这个宏带入到“示例代码 12.3.3.1”的 31 行以后就是：

```
IOMUXC_SetPinMux (0x020E0068U, 0x5U, 0x00000000U, 0x0U, 0x020E02F4U, 0);
```

这样就与函数 IOMUXC_SetPinMux 的 6 个参数对应起来了，如果我们要将 GPIO1_IO03 复用为 I2C1_SDA 的话就可以使用如下代码：

```
IOMUXC_SetPinMux(IOMUXC_GPIO1_IO03_I2C1_SDA,   0);
```

函数 IOMUXC_SetPinMux 就讲解到这里，接下来看一下函数 IOMUXC_SetPinConfig，此函数同样在文件 fsl_iomuxc.h 中有定义，函数源码如下：

```
static inline void IOMUXC_SetPinConfig(uint32_t muxRegister,
                                       uint32_t muxMode,
```

```
                                         uint32_t inputRegister,
                                         uint32_t inputDaisy,
                                         uint32_t configRegister,
                                         uint32_t configValue)
    {
        if (configRegister)
        {
            *((volatile uint32_t *)configRegister) = configValue;
        }
    }
```

函数 IOMUXC_SetPinConfig 有 6 个参数，其中前五个参数和函数 IOMUXC_SetPinMux 一样，但是此函数只使用了参数 configRegister 和 configValue，cofigRegister 参数是 IO 配置寄存器地址，比如 GPIO1_IO03 的 IO 配置寄存器为 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03，其地址为 0X020E02F4，参数 configValue 就是要写入到寄存器 configRegister 的值。同理，“示例代码 12.3.3.1”的 43 行展开以后就是：

```
IOMUXC_SetPinConfig(0x020E0068U, 0x5U, 0x00000000U, 0x0U, 0x020E02F4U, 0X10B0);
```

根据函数 IOMUXC_SetPinConfig 的源码可以知道，上面函数就是将寄存器 0x020E02F4 的值设置为 0X10B0。函数 IOMUXC_SetPinMux 和 IOMUXC_SetPinConfig 就讲解到这里，我们以后就可以使用这两个函数来方便的配置 IO 的复用功能和 IO 配置。

main.c 就讲到这里，基本和上一章一样，只是我们使用了 NXP 官方写好的寄存器定义，另外中断讲解了函数 IOMUXC_SetPinMux 和 IOMUXC_SetPinConfig。

12.4 编译下载验证

12.4.1 编写 Makefile 和链接脚本

新建 Makefile 文件，Makefile 文件内容如下：

示例代码 12.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  NAME           ?= ledc
3
4  CC             := $(CROSS_COMPILE)gcc
5  LD             := $(CROSS_COMPILE)ld
6  OBJCOPY        := $(CROSS_COMPILE)objcopy
7  OBJDUMP        := $(CROSS_COMPILE)objdump
8
9  OBJS     := start.o main.o
10
11 $(NAME).bin:$(OBJS)
12     $(LD) -Timx6ul.lds -o $(NAME).elf $^
13     $(OBJCOPY) -O binary -S $(NAME).elf $@
14     $(OBJDUMP) -D -m arm $(NAME).elf > $(NAME).dis
15
16 %.o:%.s
```

```
17      $(CC) -Wall -nostdlib -c -O2 -o $@ $<
18
19 %.o:%.S
20      $(CC) -Wall -nostdlib -c -O2 -o $@ $<
21
22 %.o:%.c
23      $(CC) -Wall -nostdlib -c -O2 -o $@ $<
24
25 clean:
26      rm -rf *.o $(NAME).bin $(NAME).elf $(NAME).dis
```

本章实验的 Makefile 文件是在第十一章中的 Makefile 上修改的，只是使用到了变量。链接脚本 imx6ul.lds 的内容和上一章一样，可以直接使用上一章的链接脚本文件。

12.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 ledc.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                  //给予 imxdownload 可执行权限，一次即可
./imxdownload ledc.bin /dev/sdd        //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板，如果代码运行正常的话 LED0 就会以 500ms 的时间间隔亮灭，实验现象和上一章一样。

第十三章 BSP 工程管理实验

在前面的章节中，我们都是将所有的源码文件刚到工程的根目录下，如果工程文件比较少的话这样做无可厚非，但是如果工程源文件达到几十、甚至数百个的时候，这样一股脑全部放到根目录下就会使工程显得混乱不堪。所以我们必须对工程文件做管理，将不同功能的源码文件放到不同的目录中。另外我们也需要将源码文件中，所有完成同一个功能的代码提取出来放到一个单独的文件中，也就是对程序分功能管理。本章我们就来学习一下如何对一个工程进行整理，使其美观、功能模块清晰、易于阅读。

13.1 工程管理简介

打开我们上一章的工程根目录，如图 13.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$ ls -a
.   cc.h         fsl_iomuxc.h  imxdownload                load.imx  Makefile     start.S
..  fsl_common.h imx6ul.lds    ledc_sdk.code-workspace    main.c    MCIMX6Y2.h   .vscode
zuozhongkai@ubuntu:~/linux/4_ledc_sdk$
```

图 13.1.1 工程根目录

在图 13.1.1 中我们将所有的源码文件都放到工程根目录下，即使这个工程只是完成了一个简单的流水灯的功能，但是其工程根目录下的源码文件就已经不少了。如果在添加一些其他的功能文件，那么文档就会更大，显得很混乱，所以我们需要对这个工程进行整理，将源码文件分模块、分功能整理。我们可以打开一个 STM32 的例程，如图 13.1.2 所示：

名称	修改日期	类型	大小
CORE	2017-12-25 12:55	文件夹	
HARDWARE	2017-12-25 12:55	文件夹	
OBJ	2017-12-25 12:55	文件夹	
STM32F10x_FWLib	2017-12-25 12:55	文件夹	
SYSTEM	2017-12-25 12:55	文件夹	
USER	2017-12-25 12:55	文件夹	
keilkilll.bat	2011-04-23 10:24	Windows 批处理文件	1 KB
README.TXT	2015-03-23 20:17	文本文档	2 KB

图 13.1.2 STM32F103 例程工程文件

图 13.1.2 中的工程目录就很美观、不同的功能模块文件放到不同的文件夹中，比如驱动文件就放到 HARDWARE 文件夹中，ST 的官方库就放到 STM32F10x_FWLib 文件夹中，编译产生的过程文件放到 OBJ 文件夹中。我们可以参考这个工程目录结构来整理第十二章的例程工程，新建名为“5_ledc_bsp”的文件夹，在里面新建 bsp、imx6ul、obj 和 project 这 4 个文件夹，完成以后如图 13.1.3 所示：

```
zuozhongkai@ubuntu:~/linux/5_ledc_bsp$ ls
bsp  imx6ul  obj  project
zuozhongkai@ubuntu:~/linux/5_ledc_bsp$
```

图 13.1.3 新建的工程根目录文件夹

其中 bsp 用来存放驱动文件；imx6ul 用来存放跟芯片有关的文件，比如 NXP 官方的 SDK 库文件；obj 用来存放编译生成的.o 文件；project 存放 start.S 和 main.c 文件，也就是应用文件；将十二章实验中的 cc.h、fsl_common.h、fsl_iomuxc.h 和 MCIMX6Y2.h 这四个文件拷贝到文件夹 imx6ul 中；将 start.S 和 main.c 这两个文件拷贝到文件夹 project 中。我们前面的实验中所有的驱动相关的函数都写到了 main.c 文件中，比如函数 clk_enable、led_init 和 delay，这三个函数可以分为三类：时钟驱动、LED 驱动和延时驱动。因此我们可以在 bsp 文件夹下创建三个子文件夹：clk、delay 和 led，分别用来存放时钟驱动文件、延时驱动文件和 LED 驱动文件，这样 main.c 函数就会清爽很多，程序功能模块清晰。工程文件夹都创建好了，接下来就是编写代码了，其实就是将时钟驱动、LED 驱动和延时驱动相关的函数从 main.c 中提取出来做成一个独立的驱动文件 。

13.2 硬件原理分析

本章使用到的硬件资源和第八章一样，就是一个 LED0。

13.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 5_ledc_bsp。

使用 VScode 新建工程，工程名字为“ledc_bsp”。

13.3.1 创建 imx6ul.h 文件

新建文件 imx6ul.h，然后保存到文件夹 imx6ul 中，在 imx6ul.h 中输入如下内容：

示例代码 13.3.1.1 imx6ul.h 文件代码

```
1  #ifndef __IMX6UL_H
2  #define __IMX6UL_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : imx6ul.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : 包含一些常用的头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/3 左忠凯创建
12 ***************************************************************/
13 #include "cc.h"
14 #include "MCIMX6Y2.h"
15 #include "fsl_common.h"
16 #include "fsl_iomuxc.h"
17 
18 #endif
```

文件 imx6ul.h 很简单，就是引用了一些头文件，以后我们就可以在其他文件中需要引用 imx6ul.h 就可以了。

13.3.2 编写 led 驱动代码

新建 bsp_led.h 和 bsp_led.c 两个文件，将这两个文件存放到 bsp/led 中，在 bsp_led.h 中输入输入如下内容：

示例代码 13.3.2.1 bsp_led.h 文件代码

```
1  #ifndef __BSP_LED_H
2  #define __BSP_LED_H
3  #include "imx6ul.h"
4  /***************************************************************
5  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6  文件名     : bsp_led.h
7  作者       : 左忠凯
8  版本       : V1.0
9  描述       : LED 驱动头文件。
10 其他       : 无
```

```
11 论坛       : www.openedv.com
12 日志       : 初版 V1.0 2019/1/4 左忠凯创建
13 ****************************************************************/
14 
15 #define LED0 0
16 
17 /* 函数声明 */
18 void led_init(void);
19 void led_switch(int led, int status);
20 #endif
```

bsp_led.h 的内容很简单，就是一些函数声明，在 bsp_led.c 中输入如下内容：

示例代码 13.3.2.2 bsp_led.c 文件代码

```
1  #include "bsp_led.h"
2  /***************************************************************
3  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
4  文件名     : bsp_led.c
5  作者       : 左忠凯
6  版本       : V1.0
7  描述       : LED 驱动文件。
8  其他       : 无
9  论坛       : www.openedv.com
10 日志       : 初版 V1.0 2019/1/4 左忠凯创建
11 ****************************************************************/
12 
13 /*
14  * @description    : 初始化 LED 对应的 GPIO
15  * @param          : 无
16  * @return         : 无
17  */
18 void led_init(void)
19 {
20     /* 1、初始化 IO 复用 */
21     IOMUXC_SetPinMux(IOMUXC_GPIO1_IO03_GPIO1_IO03,0);
22 
23     /* 2、、配置 GPIO1_IO03 的 IO 属性 */
24     IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO03_GPIO1_IO03,0X10B0);
25 
26     /* 3、初始化 GPIO,GPIO1_IO03 设置为输出*/
27     GPIO1->GDIR |= (1 << 3);
28 
29     /* 4、设置 GPIO1_IO03 输出低电平，打开 LED0*/
30     GPIO1->DR &= ~(1 << 3);
31 }
```

```
32
33 /*
34  * @description      : LED 控制函数，控制 LED 打开还是关闭
35  * @param - led      : 要控制的 LED 灯编号
36  * @param - status   : 0，关闭 LED0，1 打开 LED0
37  * @return           : 无
38  */
39 void led_switch(int led, int status)
40 {
41     switch(led)
42     {
43         case LED0:
44             if(status == ON)
45                 GPIO1->DR &= ~(1<<3);   /* 打开 LED0 */
46             else if(status == OFF)
47                 GPIO1->DR |= (1<<3);    /* 关闭 LED0 */
48             break;
49     }
50 }
```

bsp_led.c 里面就两个函数 led_init 和 led_switch，led_init 函数用来初始化 LED 所使用的 IO，led_switch 函数是控制 LED 灯的打开和关闭，这两个函数都很简单。

13.3.3 编写时钟驱动代码

新建 bsp_clk.h 和 bsp_clk.c 两个文件，将这两个文件存放到 bsp/clk 中，在 bsp_clk.h 中输入输入如下内容：

示例代码 13.3.3.1 bsp_clk.h 文件代码

```
1  #ifndef __BSP_CLK_H
2  #define __BSP_CLK_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_clk.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : 系统时钟驱动头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/4 左忠凯创建
12 ***************************************************************/
13
14 #include "imx6ul.h"
15
16 /* 函数声明 */
17 void clk_enable(void);
```

```
18
19 #endif
```

bsp_clk.h 很简单，在 bsp_clk.c 中输入内容：

示例代码 13.3.3.2 bsp_clk.c 文件代码

```
1  #include "bsp_clk.h"
2
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_clk.c
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : 系统时钟驱动。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/1/4 左忠凯创建
12 ***************************************************************/
13
14 /*
15  * @description    : 使能 I.MX6U 所有外设时钟
16  * @param          : 无
17  * @return         : 无
18  */
19 void clk_enable(void)
20 {
21     CCM->CCGR0 = 0XFFFFFFFF;
22     CCM->CCGR1 = 0XFFFFFFFF;
23     CCM->CCGR2 = 0XFFFFFFFF;
24     CCM->CCGR3 = 0XFFFFFFFF;
25     CCM->CCGR4 = 0XFFFFFFFF;
26     CCM->CCGR5 = 0XFFFFFFFF;
27     CCM->CCGR6 = 0XFFFFFFFF;
28 }
```

bsp_clk.c 只有一个 clk_enable 函数，用来使能所有的外设时钟。

13.3.4 编写延时驱动代码

新建 bsp_delay.h 和 bsp_delay.c 两个文件，将这两个文件存放到 bsp/delay 中，在 bsp_delay.h 中输入输入如下内容：

示例代码 13.3.4.1 bsp_delay.h 文件代码

```
1  #ifndef __BSP_DELAY_H
2  #define __BSP_DELAY_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_delay.h
```

```
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : 延时头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/4 左忠凯创建
12 ****************************************************************/
13 #include "imx6ul.h"
14
15 /* 函数声明 */
16 void delay(volatile unsigned int n);
17
18 #endif
```

在 bsp_delay.c 中输入内容：

示例代码 13.3.4.2 bsp_delay.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_delay.c
作者      : 左忠凯
版本      : V1.0
描述      : 延时文件。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_delay.h"
2
3  /*
4   * @description    : 短时间延时函数
5   * @param - n      : 要延时循环次数(空操作循环次数，模式延时)
6   * @return         : 无
7   */
8  void delay_short(volatile unsigned int n)
9  {
10     while(n--){}
11 }
12
13 /*
14  * @description    : 延时函数,在 396Mhz 的主频下
15  *                    延时时间大约为 1ms
16  * @param - n      : 要延时的 ms 数
17  * @return         : 无
18  */
```

```
19 void delay(volatile unsigned int n)
20 {
21     while(n--)
22     {
23         delay_short(0x7ff);
24     }
25 }
```

bsp_delay.c 里面就两个函数，delay_short 和 delay，这两个其实就是第十二章中 main.c 里面的函数。

13.3.5 修改 main.c 文件

在第十二章中，led 驱动、延时驱动和时钟驱动相关的函数全部都写到了 main.c 中，本章我们在前几节已经将这些驱动根据功能模块放置到相应的地方，所以 main.c 里面的内容就得修改，将 main.c 里面的内容改为如下所示代码：

示例代码 13.3.5.1 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 5  BSP 形式的 LED 驱动
其他      : 本实验学习目的:
            1、将各个不同的文件进行分类，学习如何整理工程、就和学习 STM32 一样创建工程
               的各个文件夹分类，实现工程文件的分类化和模块化，便于管理。
            2、深入学习 Makefile，学习 Makefile 的高级技巧，学习编写通用 Makefile。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4
5  /*
6   * @description    : mian 函数
7   * @param          : 无
8   * @return         : 无
9   */
10 int main(void)
11 {
12     clk_enable();       /* 使能所有的时钟   */
13     led_init();         /* 初始化 led      */
14
15     while(1)
```

```
16      {
17          /* 打开 LED0 */
18          led_switch(LED0,ON);
19          delay(500);
20
21          /* 关闭 LED0 */
22          led_switch(LED0,OFF);
23          delay(500);
24      }
25
26      return 0;
27 }
```

在 main.c 中我们仅仅留下了 main 函数，至此，本例程跟程序相关的内容就全部编写好了。

13.4 编译下载验证

13.4.1 编写 Makefile 和链接脚本

在工程根目录下新建 Makefile 和 imx6ul.lds 这两个文件，创建完成以后的工程如图 13.4.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/5_ledc_bsp$ ls -a
.  ..  bsp  imx6ul  imx6ul.lds  ledc_bsp.code-workspace  Makefile  obj  project  .vscode
zuozhongkai@ubuntu:~/linux/5_ledc_bsp$
```

图 13.4.1 最终的工程目录

在文件 Makefile 中输入如下所示内容：

示例代码 13.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET          ?= bsp
3
4  CC              := $(CROSS_COMPILE)gcc
5  LD              := $(CROSS_COMPILE)ld
6  OBJCOPY         := $(CROSS_COMPILE)objcopy
7  OBJDUMP         := $(CROSS_COMPILE)objdump
8
9  INCDIRS         := imx6ul \
10                     bsp/clk \
11                     bsp/led \
12                     bsp/delay
13
14 SRCDIRS         := project \
15                     bsp/clk \
16                     bsp/led \
17                     bsp/delay
18
19 INCLUDE         := $(patsubst %, -I %, $(INCDIRS))
```

```
20
21 SFILES         := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/*.S))
22 CFILES         := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/*.c))
23
24 SFILENDIR      := $(notdir  $(SFILES))
25 CFILENDIR      := $(notdir  $(CFILES))
26
27 SOBJS          := $(patsubst %, obj/%, $(SFILENDIR:.S=.o))
28 COBJS          := $(patsubst %, obj/%, $(CFILENDIR:.c=.o))
29 OBJS           := $(SOBJS) $(COBJS)
30
31 VPATH          := $(SRCDIRS)
32
33 .PHONY: clean
34
35 $(TARGET).bin : $(OBJS)
36     $(LD) -Timx6ul.lds -o $(TARGET).elf $^
37     $(OBJCOPY) -O binary -S $(TARGET).elf $@
38     $(OBJDUMP) -D -m arm $(TARGET).elf > $(TARGET).dis
39
40 $(SOBJS) : obj/%.o : %.S
41     $(CC) -Wall -nostdlib -c -O2  $(INCLUDE) -o $@ $<
42
43 $(COBJS) : obj/%.o : %.c
44     $(CC) -Wall -nostdlib -c -O2  $(INCLUDE) -o $@ $<
45
46 clean:
47  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

可以看出本章实验的 Makefile 文件要比前面的实验复杂很多，因为"示例代码 13.4.1.1"中的 Makefile 代码是一个通用 Makefile，我们以后所有的裸机例程都使用这个 Makefile。使用时候只要将所需要编译的源文件所在的目录添加到 Makefile 中即可，我们接下来详细分析一下"示例代码 13.4.1.1"中的 Makefile 源码：

第 1~7 行定义了一些变量，除了第 2 行以外其它的都是跟编译器有关的，如果使用其它编译器的话只需要修改第 1 行即可。第 2 行的变量 TARGET 目标名字，不同的例程肯定名字不一一样。

第 9 行的变量 INCDIRS 包含整个工程的.h 头文件目录，文件中的所有头文件目录都要添加到变量 INCDIRS 中。比如本例程中包含.h 头文件的目录有 imx6ul、bsp/clk、bsp/delay 和 bsp/led，所以就需要在变量 INCDIRS 中添加这些目录，即:

```
INCDIRS := imx6ul bsp/clk bsp/led bsp/delay
```

仔细观察的话会发现第 9~11 行后面都会有一个符号"\"，这个相当于"换行符"，表示本行和下一行属于同一行，一般一行写不下的时候就用符号"\"来换行。在后面的裸机例程中我们会根据实际情况来在变量 INCDIRS 中添加头文件目录。

第 14 行是变量 SRCDIRS，和变量 INCDIRS 一样，只是 SRCDIRS 包含的是整个工程的所有.c 和.S 文件目录。比如本例程包含有.c 和.S 的目录有 bsp/clk、bsp/delay、bsp/led 和 project，即：

```
SRCDIRS := project bsp/clk bsp/led bsp/delay
```

同样的，后面的裸机例程中我们也要根据实际情况在变量 SRCDIRS 中添加相应的文件目录。

第 19 行的变量 INCLUDE 是用到了函数 patsubst，通过函数 patsubst 给变量 INCDIRS 添加一个“-I”，即：

```
INCLUDE := -I imx6ul -I bsp/clk -I bsp/led -I bsp/delay
```

加“-I”的目的是因为 Makefile 语法要求指明头文件目录的时候需要加上“-I”。

第 21 行变量 SFILES 保存工程中所有的.s 汇编文件(包含绝对路径)，变量 SRCDIRS 已经存放了工程中所有的.c 和.S 文件，所以我们只需要从里面挑出所有的.S 汇编文件即可，这里借助了函数 foreach 和函数 wildcard，最终 SFILES 如下：

```
SFILES := project/start.S
```

第 22 行变量 CFILES 和变量 SFILES 一样，只是 CFILES 保存工程中所有的.c 文件(包含绝对路径)，最终 CFILES 如下：

```
CFILES = project/main.c bsp/clk/bsp_clk.c bsp/led/bsp_led.c bsp/delay/bsp_delay.c
```

第 24 和 25 行的变量 SFILENDIR 和 CFILENDIR 包含所有的.S 汇编文件和.c 文件，相比变量 SFILES 和 CFILES，SFILENDIR 和 CFILNDIR 只是文件名，不包含文件的绝对路径。使用函数 notdir 将 SFILES 和 CFILES 中的路径去掉即可，SFILENDIR 和 CFILENDIR 如下：

```
SFILENDIR = start.S
CFILENDIR = main.c bsp_clk.c bsp_led.c bsp_delay.c
```

第 27 和 28 行的变量 SOBJS 和 COBJS 是.S 和.c 文件编译以后对应的.o 文件目录，默认所有的文件编译出来的.o 文件和源文件在同一个目录中，这里我们将所有的.o 文件都放到 obj 文件夹下，SOBJS 和 COBJS 内容如下：

```
SOBJS = obj/start.o
COBJS = obj/main.o obj/bsp_clk.o obj/bsp_led.o obj/bsp_delay.o
```

第 29 行变量 OBJS 是变量 SOBJS 和 COBJS 的集合，如下：

```
OBJS = obj/start.o obj/main.o obj/bsp_clk.o obj/bsp_led.o obj/bsp_delay.o
```

编译完成以后所有的.o 文件就全部存放到了 obj 目录下，如图 13.4.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6UL/ZERO/Board_Drivers/5_ledc_bsp/obj$ ls
bsp_clk.o  bsp_delay.o  bsp_led.o  main.o  start.o
```

图 13.4.1.1 编译完成后的 obj 文件夹

第 31 行的 VPATH 是指定搜索目录的，这里指定的搜素目录就是变量 SRCDIRS 所保存的目录，这样当编译的时候所需的.S 和.c 文件就会在 SRCDIRS 中指定的目录中查找。

第 34 行指定了一个伪目标 clean，伪目标前面讲解 Makefile 的时候已经讲解过了。

第 35~47 行就很熟悉了，前几章都已经详细的讲解过了。

“示例代码 13.4.1.1”中的 Makefile 文件内容重点工作是找到要编译哪些文件？编译的.o 文件存放到哪里？使用到的编译命令和前面实验使用的一样，其实 Makefile 的重点工作就是解决“从哪里来到哪里去的”问题，也就是找到要编译的源文件、编译结果存放到哪里？真正的编译命令很简洁。

链接脚本 imx6ul.lds 的内容和上一章一样，可以直接使用上一章的链接脚本文件。

13.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 bsp.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload bsp.bin /dev/sdd           //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板，如果代码运行正常的话 LED0 就会以 500ms 的时间间隔亮灭，实验现象和上一章一样。

第十四章 蜂鸣器试验

前几章试验中的驱动 LED 灯亮灭属于 GPIO 的输出控制，本章再巩固一下 I.MX6U 的 GPIO 输出控制，在 I.MX6U-ALPHA 开发板上有一个有源蜂鸣器，通过 IO 输出高低电平即可控制蜂鸣器的开关，本质上也属于 GPIO 的输出控制。

14.2 有源蜂鸣器简介

蜂鸣器常用于计算机、打印机、报警器、电子玩具等电子产品中，常用的蜂鸣器有两种：有源蜂鸣器和无源蜂鸣器，这里的有“源”不是电源，而是震荡源，有源蜂鸣器内部带有震荡源，所以有源蜂鸣器只要通电就会叫。无源蜂鸣器内部不带震荡源，直接用直流电是驱动不起来的，需要 2K-5K 的方波去驱动。I.MX6U-ALPHA 开发板使用的是有源蜂鸣器，因此只要给其供电就会工作，I.MX6U-ALPHA 开发板所使用的有源蜂鸣器如图 14.2.1 所示：



图 14.2.1 有源蜂鸣器

有源蜂鸣器只要通电就会叫，所以我们可以做一个供电电路，这个供电电路可以由一个 IO 来控制其通断，一般使用三极管来搭建这个电路。为什么我们不能像控制 LED 灯一样，直接将 GPIO 接到蜂鸣器的负极，通过 IO 输出高低来控制蜂鸣器的通断。因为蜂鸣器工作的电流比 LED 灯要大，直接将蜂鸣器接到 I.MX6U 的 GPIO 上有可能会烧毁 IO，所以我们需要通过一个三极管来间接的控制蜂鸣器的通断，相当于加了一层隔离。本章我们就驱动 I.MX6U-ALPHA 开发板上的有源蜂鸣器，使其周期性的“滴、滴、滴.....”鸣叫。

14.3 硬件原理分析

蜂鸣器的硬件原理图如图 14.3.1 所示：

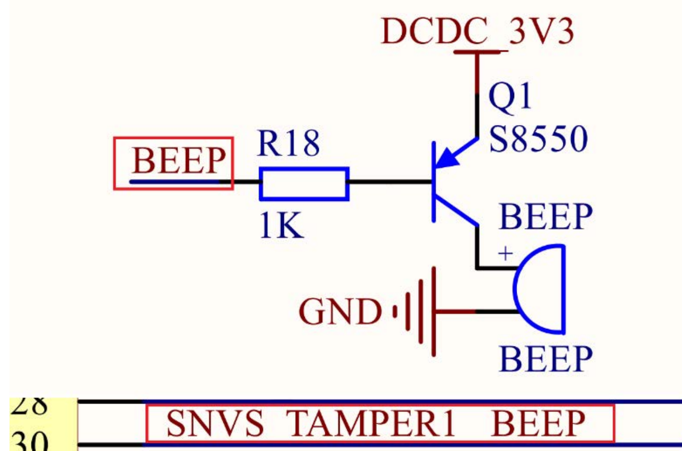


图 14.3.1 蜂鸣器原理图

图 14.3.1 中通过一个 PNP 型的三极管 8550 来驱动蜂鸣器，通过 SNVS_TAMPER1 这个 IO 来控制三极管 Q1 的导通，当 SNVS_TAMPER1 输出低电平的时候 Q1 导通，相当于蜂鸣器的正极连接到 DCDC_3V3，蜂鸣器形成一个通路，因此蜂鸣器会鸣叫。同理，当 SNVS_TAMPER1 输出高电平的时候 Q2 不导通，那么蜂鸣器就没有形成一个通路，因此蜂鸣器也就不会鸣叫。

14.3 试验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 6_beep。

新建文件夹“6_beep”，然后将上一章试验中的所有内容拷贝到刚刚新建的“6_beep”里面，拷贝完成以后的工程如图 13.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/6_beep$ ls -a
.  ..  bsp  imx6ul  imx6ul.lds  imxdownload  load.imx  Makefile  obj  project  .vscode
zuozhongkai@ubuntu:~/linux/6_beep$
```

图 13.3.1 工程文件夹

新建 VSCode 工程，工程创建完成以后在 bsp 文件夹下新建名为“beep”的文件夹，蜂鸣器驱动文件都放到“beep”文件夹里面。

新建 beep.h 文件，保存到 bsp/beep 文件夹里面，在 beep.h 里面输入如下内容：

示例代码 13.3.1 beep.h 文件代码

```
1 #ifndef __BSP_BEEP_H
2 #define __BSP_BEEP_H
3
4 #include "imx6ul.h"
5
6 /* 函数声明 */
7 void beep_init(void);
8 void beep_switch(int status);
9 #endif
```

beep.h 很简单，就是函数声明。新建文件 beep.c，然后在 beep.c 里面输入如下内容：

示例代码 13.3.2 beep.c 文件代码

```
1  #include "bsp_beep.h"
2
3  /*
4   * @description  : 初始化蜂鸣器对应的 IO
5   * @param        : 无
6   * @return       : 无
7   */
8  void beep_init(void)
9  {
10     /* 1、初始化 IO 复用，复用为 GPIO5_IO01 */
11     IOMUXC_SetPinMux(IOMUXC_SNVS_SNVS_TAMPER1_GPIO5_IO01,0);
12
13     /* 2、、配置 GPIO1_IO03 的 IO 属性   */
14     IOMUXC_SetPinConfig(IOMUXC_SNVS_SNVS_TAMPER1_GPIO5_IO01,0X10B0);
15
16     /* 3、初始化 GPIO,GPIO5_IO01 设置为输出 */
17     GPIO5->GDIR |= (1 << 1);
18
19     /* 4、设置 GPIO5_IO01 输出高电平，关闭蜂鸣器 */
20     GPIO5->DR |= (1 << 1);
```

```
21 }
22
23 /*
24  * @description      : 蜂鸣器控制函数，控制蜂鸣器打开还是关闭
25  * @param - status    : 0，关闭蜂鸣器，1 打开蜂鸣器
26  * @return           : 无
27  */
28 void beep_switch(int status)
29 {
30     if(status == ON)
31         GPIO5->DR &= ~(1 << 1); /* 打开蜂鸣器 */
32     else if(status == OFF)
33         GPIO5->DR |= (1 << 1);  /* 关闭蜂鸣器 */
}
```

beep.c 文件一共有两个函数：beep_init 和 beep_switch，其中 beep_init 用来初始化 BEEP 所使用的 GPIO，也就是 SNVS_TAMPER1，将其复用为 GPIO5_IO01，和上一章的 LED 灯初始化函数一样。beep_switch 函数用来控制 BEEP 的开关，也就是设置 GPIO5_IO01 的高低电平，很简单。

最后在 main.c 函数中输入如下所示内容：

示例代码 13.3.3 main.c 文件代码

```
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5
6  /*
7   * @description  : main 函数
8   * @param        : 无
9   * @return       : 无
10  */
11 int main(void)
12 {
13     clk_enable();          /* 使能所有的时钟    */
14     led_init();            /* 初始化 led        */
15     beep_init();           /* 初始化 beep       */
16
17  while(1)
18  {
19     /* 打开 LED0 和蜂鸣器 */
20     led_switch(LED0,ON);
21     beep_switch(ON);
22     delay(500);
23
```

```
24      /* 关闭 LED0 和蜂鸣器 */
25      led_switch(LED0,OFF);
26      beep_switch(OFF);
27      delay(500);
28  }
29
30  return 0;
31 }
```

main.c 中只有一个 main 函数，main 函数先使能所有的外设时钟，然后初始化 LED 和 BEEP。最终在 while(1)循环中周期性的开关 LED 灯和蜂鸣器，周期大约为 500ms，main.c 的内容也比较简单。

14.4 编译下载验证

14.4.1 编写 Makefile 和链接脚本

Makefile 使用第十三章编写的通用 Makefile，修改变量 TARGET 为 beep，在变量 INCDIRS 和 SRCDIRS 中追加“bsp/beep”，修改完成以后如下所示：

示例代码 13.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= beep
3
4  /* 省略掉其它代码...... */
5
6  INCDIRS        :=  imx6ul \
7                     bsp/clk \
8                     bsp/led \
9                     bsp/delay  \
10                    bsp/beep
11
12 SRCDIRS        :=  project \
13                    bsp/clk \
14                    bsp/led \
15                    bsp/delay \
16                    bsp/beep
17
18 /* 省略掉其它代码...... */
19
20 clean:
21  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改目标的名称为“beep”。

第 10 行在变量 INCDIRS 中添加蜂鸣器驱动头文件路径，也就是文件 beep.h 的路径。

第 16 行在变量 SRCDIRS 中添加蜂鸣器驱动文件路劲，也就是文件 beep.c 的路径。

链接脚本就使用第十三章试验中的链接脚本文件 imx6ul.lds 即可。

14.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 beep.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload beep.bin /dev/sdd      //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。如果代码运行正常的话 LED 灯亮的时候蜂鸣器鸣叫，当 LED 灯灭的时候蜂鸣器不鸣叫，周期大概为 500ms。通过本章的例程，我们进一步巩固了 I.MX6U 的 IO 输出控制，下一章我们学习如何实现 I.MX6U 的 IO 输入控制。

第十五章 按键输入试验

前面几章试验都是讲解如何使用 I.MX6U 的 GPIO 输出控制功能，I.MX6U 的 IO 不仅能作为输出，而且也可以作为输入。I.MX6U-ALPHA 开发板上有一个按键，按键连接了一个 IO，将这个 IO 配置为输入功能，读取这个 IO 的值即可获取按键的状态(按下或松开)。本章通过这个按键来控制蜂鸣器的开关，通过本章的学习你将掌握如何将 I.MX6UL 的 IO 作为输入来使用。

15.1 按键输入简介

按键就两个状态：按下或弹起，将按键连接到一个 IO 上，通过读取这个 IO 的值就知道按键是按下的还是弹起的。至于按键按下的时候是高电平还是低电平要根据实际电路来判断。前面几章我们都是讲解 I.MX6U 的 GPIO 作为输出使用，当 GPIO 连接按键的时候就要做为输入使用。关于 I.MX6U 的 GPIO 已经在第八章详细的讲解了，本章我们的主要工作就是配置按键所连接的 IO 为输入功能，然后读取这个 IO 的值来判断按键是否按下。

I.MX6U-ALPHA 开发板上有一个按键 KEY0，本章我们将会编写代码通过这个 KEY0 按键来控制开发板上的蜂鸣器，按一下 KEY0 蜂鸣器打开，再按一下蜂鸣器就关闭。

15.2 硬件原理分析

本试验我们用到的硬件有：

1） LED 灯 LED0。

2）蜂鸣器。

3）1 个按键 KEY0。

按键 KEY0 的原理图如图 15.2.1 所示：

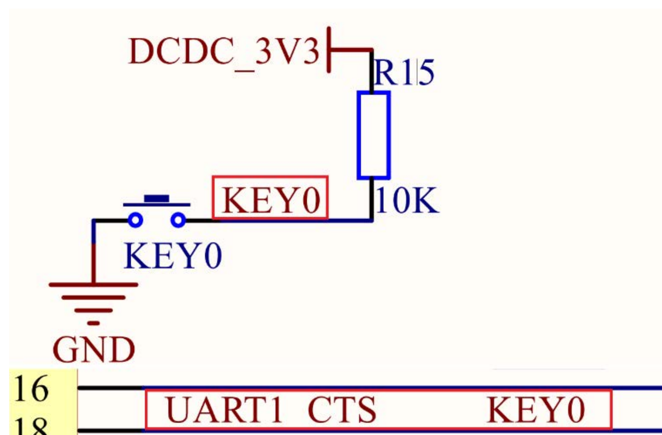


图 15.2.1 按键原理图

从图 15.2.1 可以看出，按键 KEY0 是连接到 I.MX6U 的 UART1_CTS 这个 IO 上的，KEY0 接了一个 10K 的上拉电阻，因此 KEY0 没有按下的时候 UART1_CTS 应该是高电平，当 KEY0 按下以后 UART1_CTS 就是低电平。

15.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 7_key。

本试验在上一章试验例程的基础上完成，重新创建 VSCode 工程，工作区名字为“key”，在工程目录的 bsp 文件夹中创建名为“key”和“gpio”两个文件夹。按键相关的驱动文件都放到“key”文件夹中，本章试验我们对 GPIO 的操作编写一个函数集合，也就是编写一个 GPIO 驱动文件，GPIO 的驱动文件放到“gpio”文件夹里面。

新建 bsp_gpio.c 和 bsp_gpio.h 这两个文件，将这两个文件都保存到刚刚创建的 bsp/gpio 文件夹里面，然后在 bsp_gpio.h 文件夹里面输入如下内容：

示例代码 15.3.1 bsp_gpio.h 文件代码

```
1  #ifndef _BSP_GPIO_H
2  #define _BSP_GPIO_H
```

```
3  #define _BSP_KEY_H
4  #include "imx6ul.h"
5  /***************************************************************
6  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
7  文件名    : bsp_gpio.h
8  作者      : 左忠凯
9  版本      : V1.0
10 描述      : GPIO 操作文件头文件。
11 其他      : 无
12 论坛      : www.openedv.com
13 日志      : 初版 V1.0 2019/1/4 左忠凯创建
14 ***************************************************************/
15 
16 /* 枚举类型和结构体定义 */
17 typedef enum _gpio_pin_direction
18 {
19     kGPIO_DigitalInput = 0U,        /* 输入 */
20     kGPIO_DigitalOutput = 1U,       /* 输出 */
21 } gpio_pin_direction_t;
22 
23 /* GPIO 配置结构体 */
24 typedef struct _gpio_pin_config
25 {
26     gpio_pin_direction_t direction; /* GPIO 方向:输入还是输出      */
27     uint8_t outputLogic;            /* 如果是输出的话，默认输出电平  */
28 } gpio_pin_config_t;
29 
30 
31 /* 函数声明 */
32 void gpio_init(GPIO_Type *base, int pin, gpio_pin_config_t *config);
33 int gpio_pinread(GPIO_Type *base, int pin);
34 void gpio_pinwrite(GPIO_Type *base, int pin, int value);
35 
36 #endif
```

bsp_gpio.h 中定义了一个枚举类型 gpio_pin_direction_t 和结构体 gpio_pin_config_t，枚举类型 gpio_pin_direction_t 表示 GPIO 方向，输入或输出。结构体 gpio_pin_config_t 是 GPIO 的配置结构体，里面有 GPIO 的方向和默认输出电平两个成员变量。在 bsp_gpio.c 中输入如下所示内容：

示例代码 15.3.2 bsp_gpio.c 文件代码

```
1  #include "bsp_gpio.h"
2  /***************************************************************
3  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
4  文件名    : bsp_gpio.h
```

```
5  作者       : 左忠凯
6  版本       : V1.0
7  描述       : GPIO 操作文件。
8  其他       : 无
9  论坛       : www.openedv.com
10 日志       : 初版 V1.0 2019/1/4 左忠凯创建
11 ***************************************************************/
12 
13 /*
14  * @description    : GPIO 初始化。
15  * @param - base   : 要初始化的 GPIO 组。
16  * @param - pin    : 要初始化 GPIO 在组内的编号。
17  * @param - config : GPIO 配置结构体。
18  * @return          : 无
19  */
20 void gpio_init(GPIO_Type *base, int pin, gpio_pin_config_t *config)
21 {
22     if(config->direction == kGPIO_DigitalInput)   /* 输入 */
23     {
24         base->GDIR &= ~( 1 << pin);
25     }
26     else                                           /* 输出 */
27     {
28         base->GDIR |= 1 << pin;
29         gpio_pinwrite(base,pin, config->outputLogic);/* 默认输出电平 */
30     }
31 }
32 
33  /*
34   * @description   : 读取指定 GPIO 的电平值 。
35   * @param – base  : 要读取的 GPIO 组。
36   * @param - pin   : 要读取的 GPIO 脚号。
37   * @return        : 无
38   */
39  int gpio_pinread(GPIO_Type *base, int pin)
40  {
41     return (((base->DR) >> pin) & 0x1);
42  }
43 
44  /*
45   * @description   : 指定 GPIO 输出高或者低电平 。
46   * @param – base  : 要输出的的 GPIO 组。
47   * @param - pin   : 要输出的 GPIO 脚号。
```

```
48  * @param - value : 要输出的电平，1 输出高电平， 0 输出低低电平
49  * @return        : 无
50  */
51 void gpio_pinwrite(GPIO_Type *base, int pin, int value)
52 {
53     if (value == 0U)
54     {
55         base->DR &= ~(1U << pin); /* 输出低电平 */
56     }
57     else
58     {
59         base->DR |= (1U << pin); /* 输出高电平 */
60     }
61 }
```

文件 bsp_gpio.c 中有三个函数：gpio_init、gpio_pinread 和 gpio_pinwrite，函数 gpio_init 用于初始化指定的 GPIO 引脚，最终配置的是 GDIR 寄存器，此函数有三个参数，这三个参数的含义如下：

base: 要初始化的 GPIO 所属于的 GPIO 组，比如 GPIO1_IO18 就属于 GPIO1 组。

pin: 要初始化 GPIO 在组内的标号，比如 GPIO1_IO18 在组内的编号就是 18。

config: 要初始化的 GPIO 配置结构体，用来指定 GPIO 配置为输出还是输入。

函数 gpio_pinread 是读取指定的 GPIO 值，也就是读取 DR 寄存器的指定位，此函数有两个参数和一个返回值，参数含义如下：

base: 要读取的 GPIO 所属于的 GPIO 组，比如 GPIO1_IO18 就属于 GPIO1 组。

pin: 要读取的 GPIO 在组内的标号，比如 GPIO1_IO18 在组内的编号就是 18。

返回值: 读取到的 GPIO 值，为 0 或者 1。

函数 gpio_pinwrite 是控制指定的 GPIO 引脚输入高电平(1)或者低电平(0)，就是设置 DR 寄存器的指定位，此函数有三个参数，参数含义如下：

base: 要设置的 GPIO 所属于的 GPIO 组，比如 GPIO1_IO18 就属于 GPIO1 组。

pin: 要设置的 GPIO 在组内的标号，比如 GPIO1_IO18 在组内的编号就是 18。

value: 要设置的值，1(高电平)或者 0(低电平)。

我们以后就可以使用函数 gpio_init 设置指定 GPIO 为输入还是输出，使用函数 gpio_pinread 和 gpio_pinwrite 来读写指定的 GPIO，文件 bsp_gpio.c 文件就讲解到这里。

接下来编写按键驱动文件，新建 bsp_key.c 和 bsp_key.h 这两个文件，将这两个文件都保存到刚刚创建的 bsp/key 文件夹里面，然后在 bsp_key.h 文件夹里面输入如下内容：

示例代码 15.3.3 bsp_key.h 文件代码

```
1 #ifndef _BSP_KEY_H
2 #define _BSP_KEY_H
3 #include "imx6ul.h"
4 /***************************************************************
5 Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6 文件名    : bsp_key.h
7 作者      : 左忠凯
8 版本      : V1.0
```

```
9  描述       : 按键驱动头文件。
10 其他       : 无
11 论坛       : www.openedv.com
12 日志       : 初版 V1.0 2019/1/4 左忠凯创建
13 ***************************************************************/
14 
15 /* 定义按键值 */
16 enum keyvalue{
17     KEY_NONE   = 0,
18     KEY0_VALUE,
19 };
20 
21 /* 函数声明 */
22 void key_init(void);
23 int key_getvalue(void);
24 
25 #endif
```

bsp_key.h 文件中定义了一个枚举类型：keyvalue，此枚举类型表示按键值，因为 I.MX6U-ALPHA 开发板上只有一个按键，因此枚举类型里面只到 KEY0_VALUE。在 bsp_key.c 中输入如下所示内容：

示例代码 15.3.4 bsp_key.c 文件代码

```
1  #include "bsp_key.h"
2  #include "bsp_gpio.h"
3  #include "bsp_delay.h"
4  /***************************************************************
5  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6  文件名      : bsp_key.c
7  作者       : 左忠凯
8  版本       : V1.0
9  描述       : 按键驱动文件。
10 其他       : 无
11 论坛       : www.openedv.com
12 日志       : 初版 V1.0 2019/1/4 左忠凯创建
13 ***************************************************************/
14 
15 /*
16  * @description    : 初始化按键
17  * @param          : 无
18  * @return         : 无
19  */
20 void key_init(void)
21 {
22     gpio_pin_config_t key_config;
```

```
23
24      /* 1、初始化 IO 复用, 复用为 GPIO1_IO18 */
25      IOMUXC_SetPinMux(IOMUXC_UART1_CTS_B_GPIO1_IO18, 0);
26
27      /* 2、、配置 UART1_CTS_B 的 IO 属性
28       *bit 16:0 HYS 关闭
29       *bit [15:14]: 11 默认 22K 上拉
30       *bit [13]: 1 pull 功能
31       *bit [12]: 1 pull/keeper 使能
32       *bit [11]: 0 关闭开路输出
33       *bit [7:6]: 10 速度 100Mhz
34       *bit [5:3]: 000 关闭输出
35       *bit [0]: 0 低转换率
36      */
37      IOMUXC_SetPinConfig(IOMUXC_UART1_CTS_B_GPIO1_IO18, 0xF080);
38
39      /* 3、初始化 GPIO GPIO1_IO18 设置为输入*/
40      key_config.direction = kGPIO_DigitalInput;
41      gpio_init(GPIO1,18, &key_config);
42
43 }
44
45 /*
46  * @description    : 获取按键值
47  * @param          : 无
48  * @return         : 0 没有按键按下，其他值:对应的按键值
49  */
50 int key_getvalue(void)
51 {
52      int ret = 0;
53      static unsigned char release = 1;  /* 按键松开 */
54
55      if((release==1)&&(gpio_pinread(GPIO1, 18) == 0)) /* KEY0 按下 */
56      {
57          delay(10);      /* 延时消抖    */
58          release = 0;    /* 标记按键按下 */
59          if(gpio_pinread(GPIO1, 18) == 0)
60              ret = KEY0_VALUE;
61      }
62      else if(gpio_pinread(GPIO1, 18) == 1)     /* KEY0 未按下 */
63      {
64          ret = 0;
65          release = 1;    /* 标记按键释放 */
```

```
66     }
67 
68     return ret;
69 }
```

bsp_key.c 中一共有两个函数：key_init 和 key_getvalue，key_init 是按键初始化函数，用来初始化按键所使用的 UART1_CTS 这个 IO。函数 key_init 先设置 UART1_CTS 复用为 GPIO1_IO18，然后配置 UART1_CTS 这个 IO 为速度为 100MHz，默认 22K 上拉。最后调用函数 gpio_init 来设置 GPIO1_IO18 为输入功能。

函数 key_getvalue 用于获取按键值，此函数没有参数，只有一个返回值，返回值表示按键值，返回值为 0 的话就表示没有按键按下，如果返回其他值的话就表示对应的按键按下了。获取按键值其实就是不断的读取 GPIO1_IO18 的值，如果按键按下的话相应的 IO 被拉低，那么 GPIO1_IO18 值就为 0，如果按键未按下的话 GPIO1_IO18 的值就为 1。此函数中静态局部变量 release 表示按键是否释放。

"示例代码 15.3.4"中的 57 行是按键消抖延时函数，延时时间大约为 10ms，用于消除按键抖动。理想型的按键电压变化过程如图 15.3.1 所示：

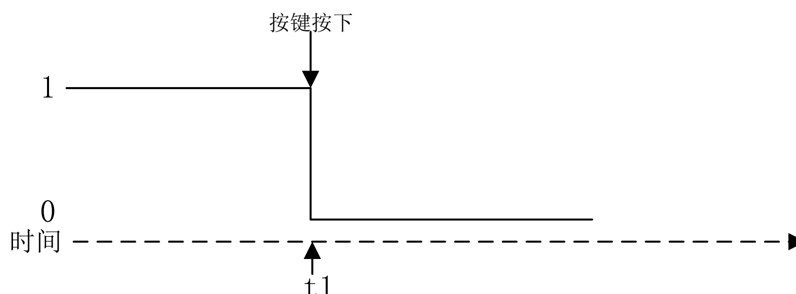


图 15.3.1 理想的按键电压变化过程

在图 15.3.1 中，按键没有按下的时候按键值为 1，当按键在 t1 时刻按键被按下以后按键值就变为 0，这是最理想的状态。但是实际的按键是机械结构，加上刚按下去的一瞬间人手可能也有抖动，实际的按键电压变化过程如图 15.3.2 所示：

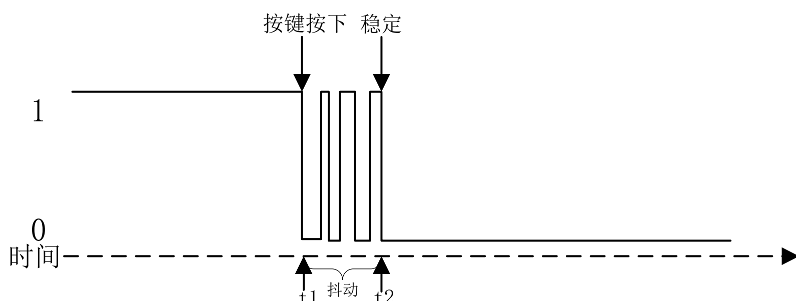


图 15.3.2 实际的按键电压变化过程

在图 15.3.2 中 t1 时刻按键被按下，但是由于抖动的原因，直到 t2 时刻才稳定下来，t1 到 t2 这段时间就是抖动。一般这段时间就是十几 ms 左右，从图 15.3.2 中可以看出在抖动期间会有多次触发，如果不消除这段抖动的话软件就会误判，本来按键就按下了一次，结果软件读取 IO 值发现电平多次跳变以为按下了多次。所以我们需要跳过这段抖动时间再去读取按键的 IO 值，也就是至少要在 t2 时刻以后再去读 IO 值。在"示例代码 15.3.4"中的 57 行是延时了大约 10ms 后再去读取 GPIO1_IO18 的 IO 值，如果此时按键的值依旧是 0，那么就表示这是一次有效的按键触发。

按键驱动就讲解到这里，最后就是 main.c 文件的内容了，在 main.c 中输入如下代码：

示例代码 15.3.5 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   :   mian.c
作者     : 左忠凯
版本     : V1.0
描述     : I.MX6U 开发板裸机实验 7 按键输入实验
其他     : 本实验主要学习如何配置 I.MX6U 的 GPIO 作为输入来使用，通过
           开发板上的按键控制蜂鸣器的开关。
论坛     : www.openedv.com
日志     : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6
7  /*
8   * @description    : main 函数
9   * @param          : 无
10  * @return         : 无
11  */
12 int main(void)
13 {
14     int i = 0;
15     int keyvalue = 0;
16     unsigned char led_state = OFF;
17     unsigned char beep_state = OFF;
18
19     clk_enable();      /* 使能所有的时钟  */
20     led_init();        /* 初始化 led      */
21     beep_init();       /* 初始化 beep     */
22     key_init();        /* 初始化 key      */
23
24     while(1)
25     {
26         keyvalue = key_getvalue();
27         if(keyvalue)
28         {
29             switch (keyvalue)
30             {
```

```
31                  case KEY0_VALUE:
32                      beep_state = !beep_state;
33                      beep_switch(beep_state);
34                      break;
35              }
36          }
37          i++;
38          if(i==50)
39          {
40              i = 0;
41              led_state = !led_state;
42              led_switch(LED0, led_state);
43          }
44          delay(10);
45      }
46      return 0;
47 }
```

main.c 函数先初始化 led 灯、蜂鸣器和按键，然后在 while(1)循环中不断的调用函数 key_getvalue 来读取按键值，如果 KEY0 按下的话就打开/关闭蜂鸣器。LED0 作为系统提示指示灯闪烁，闪烁周期大约为 500ms。本章例程的软件编写就到这里结束了，接下来就是编译下载验证了。

15.4 编译下载验证

15.4.1 编写 Makefile 和链接脚本

Makefile 使用第十三章编写的通用 Makefile，修改变量 TARGET 为 beep，在变量 INCDIRS 和 SRCDIRS 中追加“bsp/beep”，修改完成以后如下所示：

示例代码 15.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= key
3
4   /* 省略掉其它代码...... */
5
6  INCDIRS        := imx6ul \
7                     bsp/clk \
8                     bsp/led \
9                     bsp/delay  \
10                    bsp/beep \
11                    bsp/gpio \
12                    bsp/key
13
14 SRCDIRS        := project \
15                    bsp/clk \
```

```
16                    bsp/led \
17                    bsp/delay \
18                    bsp/beep \
19                    bsp/gpio \
20                    bsp/key
21
22 /* 省略掉其它代码...... */
23
24 clean:
25  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“key”，也就是目标名称为“key”。

第 11、12 行在变量 INCDIRS 中添加 GPIO 和按键驱动头文件(.h)路径。

第 19、20 行在变量 SRCDIRS 中添加 GPIO 和按键驱动文件(.c)路径。

链接脚本就使用第十三章试验中的链接脚本文件 imx6ul.lds 即可。

15.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 key.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload key.bin /dev/sdd           //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。如果代码运行正常的话 LED0 会以大约 500ms 周期闪烁， 按下开发板上的 KEY0 按键，蜂鸣器打开，再按下 KEY0 按键，蜂鸣器关闭。

第十六章 主频和时钟配置实验

在前几章实验中我们都没有涉及到 I.MX6U 的时钟和主频配置操作，全部使用的默认配置，默认配置下 I.MX6U 工作频率为 396MHz。但是 I.MX6U 系列标准的工作频率为 528MHz，有些型号甚至可以工作到 696MHz。本章我们就学习 I.MX6U 的时钟系统，学习如何配置 I.MX6U 的系统时钟和其他的外设时钟，使其工作频率为 528MHz，其他的外设时钟源都工作在 NXP 推荐的频率。

16.1 I.MX6U 时钟系统详解

I.MX6U 的系统主频为 528MHz，有些型号可以跑到 696MHz，但是默认情况下内部 boot rom 会将 I.MX6U 的主频设置为 396MHz，这个我们在 9.2 小节已经讲过了。我们在使用 I.MX6U 的时候肯定是要发挥它的最大性能，那么主频肯定要设置到 528MHz(其它型号可以设置更高，比如 696MHz)，其它的外设时钟也要设置到 NXP 推荐的值。I.MX6U 的系统时钟在《I.MX6ULL/I.MX6UL 参考手册》的第 10 章“Chapter 10 Clock and Power Management”和第 18 章“Chapter 18 Clock Controller Module (CCM)”这两章有详细的讲解。

16.1.1 系统时钟来源

打开 I.MX6U-ALPHA 开发板原理图，开发板时钟原理图如图 16.1.1.1 所示：

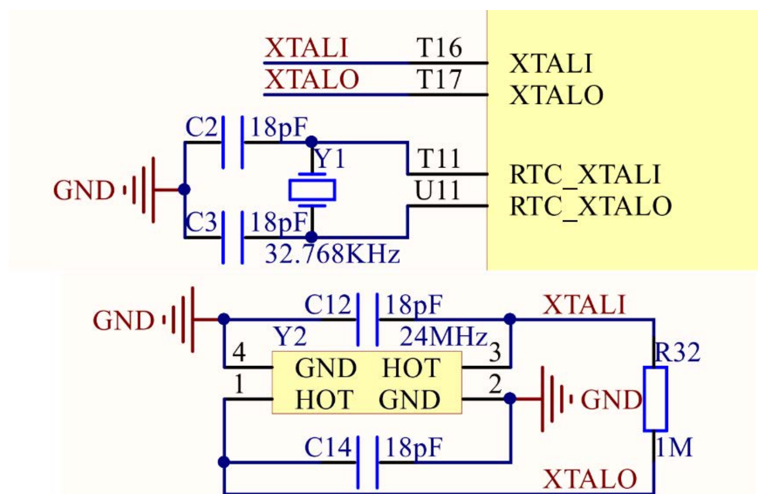


图 16.1.1.1 开发板时钟原理图

从图 16.1.1.1 可以看出 I.MX6U-ALPHA 开发板的系统时钟来源于两部分：32.768KHz 和 24MHz 的晶振，其中 32.768KHz 晶振是 I.MX6U 的 RTC 时钟源，24MHz 晶振是 I.MX6U 内核和其它外设的时钟源，也是我们重点要分析的。

16.1.2 7 路 PLL 时钟源

I.MX6U 的外设有很多，不同的外设时钟源不同，NXP 将这些外设的时钟源进行了分组，一共有 7 组，这 7 组时钟源都是从 24MHz 晶振 PLL 而来的，因此也叫做 7 组 PLL，这 7 组 PLL 结构如图 16.1.1.2 所示：

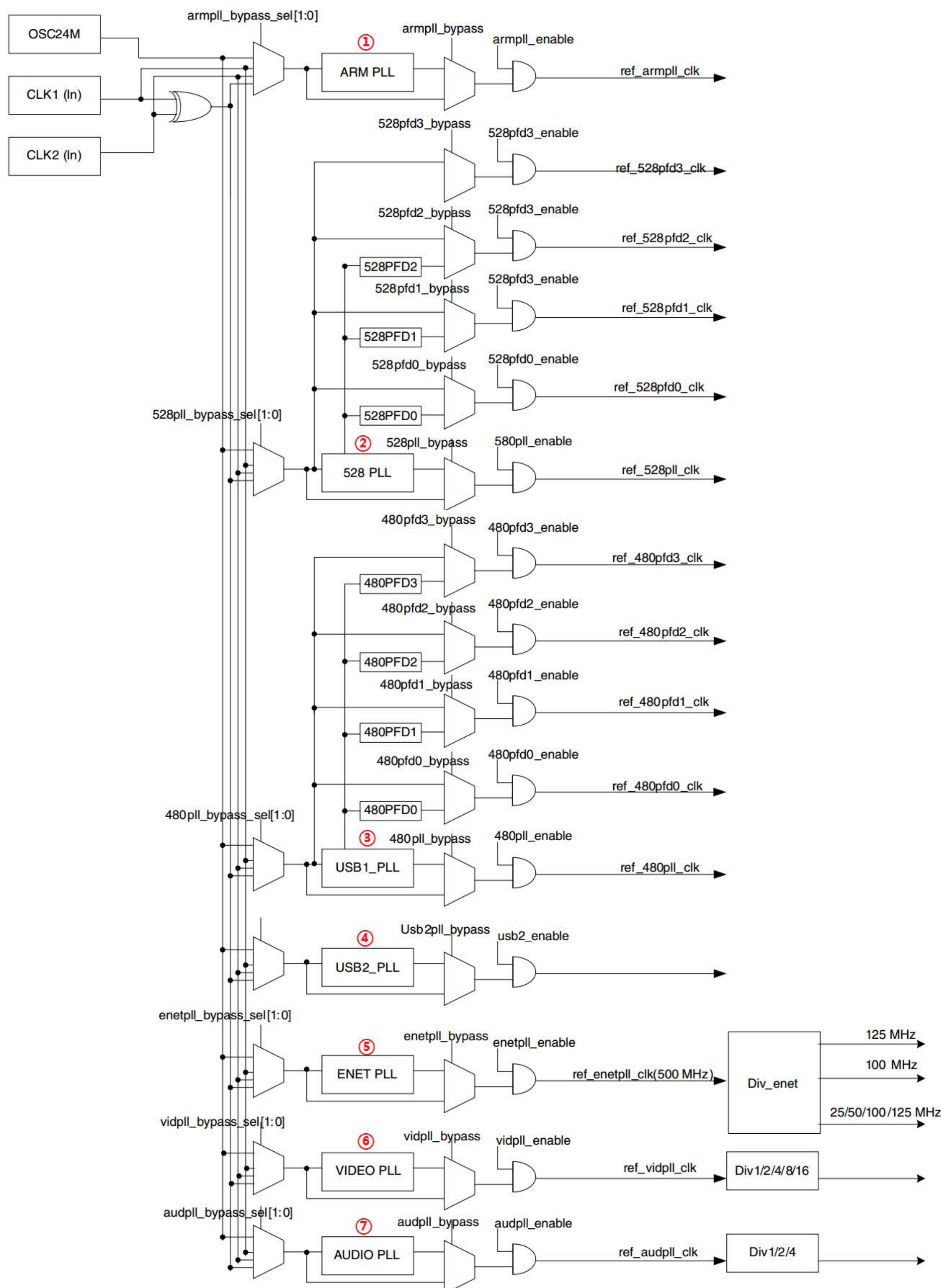


图 16.1.2.1 初级 PLLs 时钟源生成图

图 16.1.2.1 展示了 7 个 PLL 的关系，我们依次来看一下这 7 个 PLL 都是什么做什么的：

、 ARM_PLL（PLL1），此路 PLL 是供 ARM 内核使用的，ARM 内核时钟就是由此 PLL 生成的，此 PLL 通过编程的方式最高可倍频到 1.3GHz。

、528_PLL(PLL2)，此路 PLL 也叫做 System_PLL，此路 PLL 是固定的 22 倍频，不可编程

修改。因此，此路 PLL 时钟=24MHz * 22 = 528MHz，这也是为什么此 PLL 叫做 528_PLL 的原因。此 PLL 分出了 4 路 PFD，分别为：PLL2_PFD0~PLL2_PFD3，这 4 路 PFD 和 528_PLL 共同作为其它很多外设的根时钟源。通常 528_PLL 和这 4 路 PFD 是 I.MX6U 内部系统总线的时钟源，比如内处理逻辑单元、DDR 接口、NAND/NOR 接口等等。

、USB1_PLL(PLL3)，此路 PLL 主要用于 USBPHY，此 PLL 也有四路 PFD，为：PLL3_PFD0~PLL3_PFD3，USB1_PLL 是固定的 20 倍频，因此 USB1_PLL=24MHz *20=480MHz。USB1_PLL 虽然主要用于 USB1PHY，但是其和四路 PFD 同样也可以作为其他外设的根时钟源。

④、USB2_PLL(PLL7，没有写错！就是 PLL7，虽然序号标为 4，但是实际是 PLL7)，看名字就知道此路 PLL 是给 USB2PHY 使用的。同样的，此路 PLL 固定为 20 倍频，因此也是 480MHz。

⑤、ENET_PLL(PLL6),此路 PLL 固定为 20+5/6 倍频，因此 ENET_PLL=24MHz * (20+5/6) = 500MHz。此路 PLL 用于生成网络所需的时钟，可以在此 PLL 的基础上生成 25/50/100/125MHz 的网络时钟。

⑥、VIDEO_PLL(PLL5),此路 PLL 用于显示相关的外设，比如 LCD，此路 PLL 的倍频可以调整，PLL 的输出范围在 650MHz~1300MHz。此路 PLL 在最终输出的时候还可以进行分频，可选 1/2/4/8/16 分频。

⑦、AUDIO_PLL(PLL4),此路 PLL 用于音频相关的外设，此路 PLL 的倍频可以调整，PLL 的输出范围同样也是 650MHz~1300MHz，此路 PLL 在最终输出的时候也可以进行分频，可选 1/2/4 分频。

16.1.3 时钟树简介

在上一小节讲解了 7 路 PLL，I.MX6U 的所有外设时钟源都是从这 7 路 PLL 和有些 PLL 的 PFD 而来的，这些外设究竟是如何选择 PLL 或者 PFD 的？这个就要借助《IMX6ULL 参考手册》里面的时钟树了，在“Chapter 18 Clock Controller Module (CCM)”的 18.3 小节给出了 I.MX6U 详细的时钟树图，如图 16.1.3.1 所示：

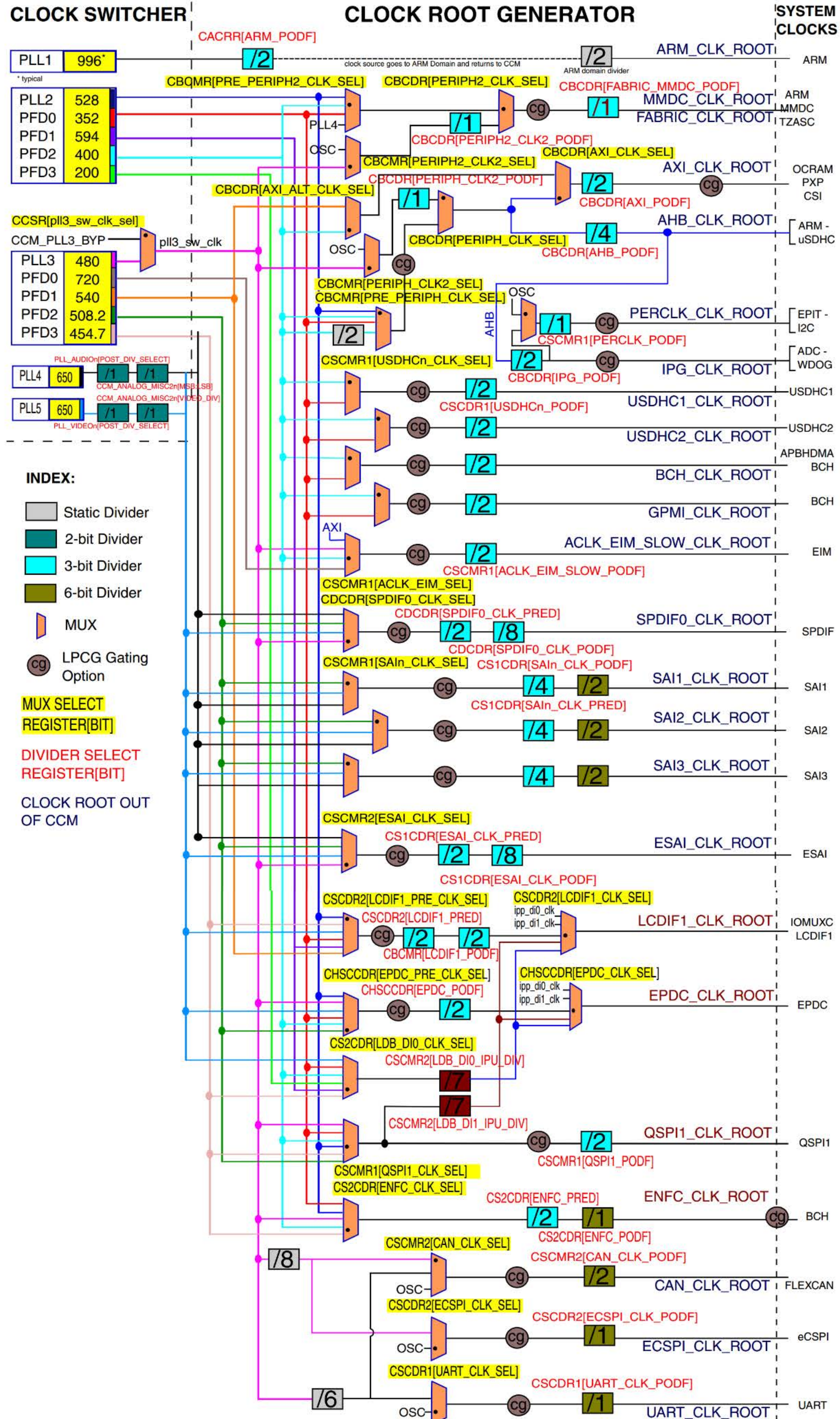

CLOCK SWITCHER
CLOCK ROOT GENERATOR
SYSTEM CLOCKS
ARM_CLK_ROOT
MMDC_CLK_ROOT
FABRIC_CLK_ROOT
AXI_CLK_ROOT
AHB_CLK_ROOT
PERCLK_CLK_ROOT
IPG_CLK_ROOT
USDHC1_CLK_ROOT
USDHC2_CLK_ROOT
BCH_CLK_ROOT
GPMI_CLK_ROOT
ACLK_EIM_SLOW_CLK_ROOT
SPDIF0_CLK_ROOT
SAI1_CLK_ROOT
SAI2_CLK_ROOT
SAI3_CLK_ROOT
ESAI_CLK_ROOT
LCDIF1_CLK_ROOT
EPDC_CLK_ROOT
QSPI1_CLK_ROOT
ENFC_CLK_ROOT
CAN_CLK_ROOT
ECSPI_CLK_ROOT
UART_CLK_ROOT
INDEX:
Static Divider
2-bit Divider
3-bit Divider
6-bit Divider
MUX
LPCG Gating Option
MUX SELECT REGISTER[BIT]
DIVIDER SELECT REGISTER[BIT]
CLOCK ROOT OUT OF CCM

图 16.1.3.1 I.MX6U 时钟树

在图 16.1.3.1 中一共有三部分：CLOCK_SWITCHER、CLOCK ROOT GENERATOR 和 SYSTEM CLOCKS。其中左边的 CLOCK_SWITCHER 就是我们上一小节讲解的那 7 路 PLL 和 8 路 PFD，右边的 SYSTEM CLOCKS 就是芯片外设，中间的 CLOCK ROOT GENERATOR 是最复杂的!这一部分就像“月老”一样，给左边的 CLOCK_SWITCHER 和右边的 SYSTEM CLOCKS 进行牵线搭桥。外设时钟源是有多路可以选择的，CLOCK ROOT GENERATOR 就负责从 7 路 PLL 和 8 路 PFD 中选择合适的时钟源给外设使用。具体操作肯定是设置相应的寄存器，我们以 ESAI 这个外设为例，ESAI 的时钟图如图 16.1.3.2 所示：

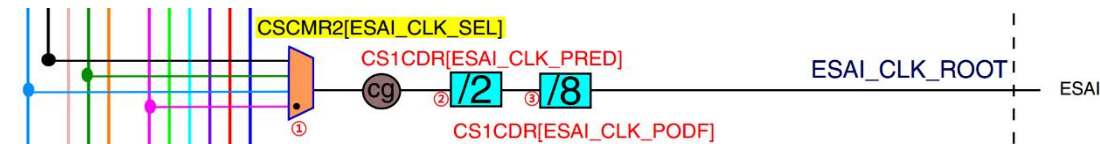


图 16.1.3.2 ESAI 时钟

在图 16.1.3.2 中我们分为了 3 部分，这三部分如下：

①、此部分是时钟源选择器，ESAI 有 4 个可选的时钟源：PLL4、PLL5、PLL3_PFD2 和 pll3_sw_clk。具体选择哪一路作为 ESAI 的时钟源是由寄存器 CCM->CSCMR2 的 ESAI_CLK_SEL 位来决定的，用户可以自由配置，配置如图 16.1.3.3 所示：

20–19 ESAI_CLK_SEL	Selector for the ESAI clock 00 derive clock from PLL4 divided clock 01 derive clock from PLL3 PFD2 clock 10 derive clock from PLL5 clock 11 derive clock from pll3_sw_clk

图 16.1.3.3 寄存器 CSCMR2 的 ESAI_CLK_SEL 位

②、此部分是 ESAI 时钟的前级分频，分频值由寄存器 CCM_CS1CDR 的 ESAI_CLK_PRED 来确定的，可设置 1~8 分频，假如现在 PLL4=650MHz，我们选择 PLL4 作为 ESAI 时钟，前级分频选择 2 分频，那么此时的时钟就是 650/2=325MHz。

③、此部分又是一个分频器，对②中输出的时钟进一步分频，分频值由寄存器 CCM_CS1CDR 的 ESAI_CLK_PODF 来决定，可设置 1~8 分频。假如我们设置为 8 分频的话，经过此分频器以后的时钟就是 325/8=40.625MHz。因此最终进入到 ESAI 外设的时钟就是 40.625MHz。

上面我们以外设 ESAI 为例讲解了如何根据图 16.1.3.1 来设置外设的时钟频率，其他的外设基本类似的，大家可以自行分析一下其他的外设。关于外设时钟配置相关内容全部都在《I.MX6ULL 参考手册》的第 18 章。

16.1.4 内核时钟设置

I.MX6U 的时钟系统前面几节已经分析的差不多了，现在就可以开始设置相应的时钟频率了。先从主频开始，我们将 I.MX6U 的主频设置为 528MHz，根据图 16.1.3.2 的时钟树可以看到 ARM 内核时钟如图 16.1.4.1 所示：

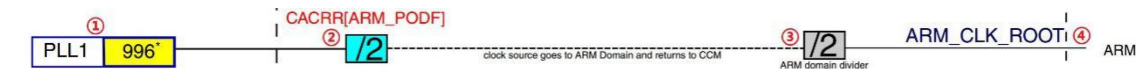


图 16.1.4.1 ARM 内核时钟树

在图 16.1.4.1 中各部分如下：

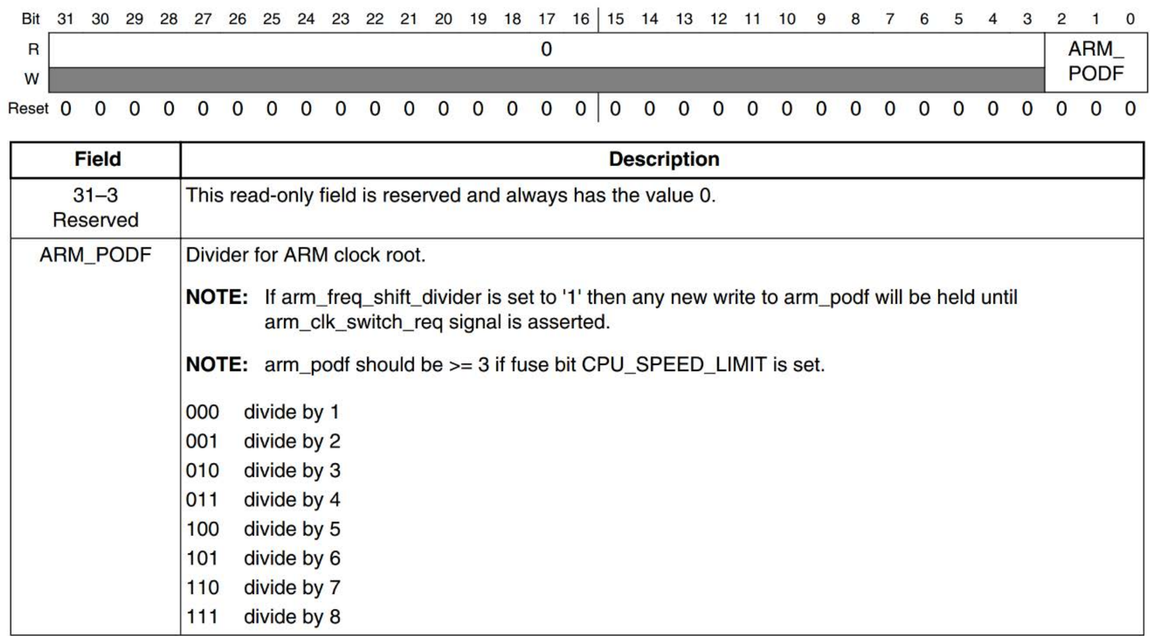
①、内核时钟源来自于 PLL1，假如此时 PLL1 为 996MHz。

②、通过寄存器 CCM_CACRR 的 ARM_PODF 位对 PLL1 进行分频，可选择 1/2/4/8 分频，假如我们选择 2 分频，那么经过分频以后的时钟频率是 996/2=498MHz。

③、大家不要被此处的 2 分频给骗了，此处没有进行 2 分频(我就被这个 2 分频骗了好久，主频一直配置不正确！)。

④、经过第②步 2 分频以后的 498MHz 就是 ARM 的内核时钟，也就是 I.MX6U 的主频。

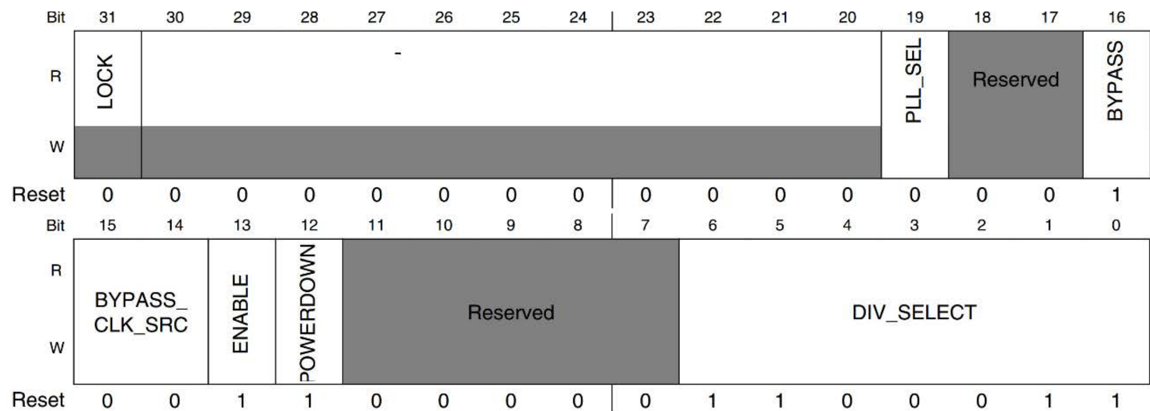
经过上面几步的分析可知，假如我们要设置内核主频为 528MHz，那么 PLL1 可以设置为 1056MHz，寄存器 CCM_CACRR 的 ARM_PODF 位设置为 2 分频即可。同理，如果要将主频设置为 696MHz，那么 PLL1 就可以设置为 696MHz，CCM_CACRR 的 ARM_PODF 设置为 1 分频即可。现在问题很清晰了，寄存器 CCM_CACCR 的 ARM_PODF 位很好设置，PLL1 的频率可以通过寄存器 CCM_ANALOG_PLL_ARMn 来设置。接下来详细的看一下 CCM_CACRR 和 CCM_ANALOG_PLL_ARMn 这两个寄存器，CCM_CACRR 寄存器结构如图 16.1.4.2 所示：



Field	Description
31–3 Reserved	This read-only field is reserved and always has the value 0.
ARM_PODF	Divider for ARM clock root. **NOTE:** If arm_freq_shift_divider is set to '1' then any new write to arm_podf will be held until arm_clk_switch_req signal is asserted. **NOTE:** arm_podf should be >= 3 if fuse bit CPU_SPEED_LIMIT is set. 000 divide by 1 001 divide by 2 010 divide by 3 011 divide by 4 100 divide by 5 101 divide by 6 110 divide by 7 111 divide by 8

图 16.1.4.2 寄存器 CCM_CACRR

寄存器 CCM_CACRR 只有 ARM_PODF 位，可以设置为 0~7，分别对应 1~8 分频。如果要设置为2分频的话CCM_CACCR就要设置为1。再来看一下寄存器CCM_ANALOG_PLL_ARMn，此寄存器结构如图 16.1.4.3 所示：



CCM_ANALOG_PLL_ARM*n* field descriptions

Field	Description
31 LOCK	1 - PLL is currently locked. 0 - PLL is not currently locked.
30–20 -	Always set to zero (0).
19 PLL_SEL	Reserved
18–17 -	This field is reserved. Reserved
16 BYPASS	Bypass the PLL.
15–14 BYPASS_CLK_SRC	Determines the bypass source. **NOTE:** Changing the Bypass clock source also changes the PLL reference clock source. 0x0 **REF_CLK_24M** — Select the 24MHz oscillator as source. 0x1 **CLK1** — Select the CLK1_N / CLK1_P as source. 0x2 **Reserved** — 0x3 **Reserved** —
13 ENABLE	Enable the clock output.
12 POWERDOWN	Powers down the PLL.
11–7 -	This field is reserved. Reserved.
DIV_SELECT	This field controls the PLL loop divider. Valid range for divider value: 54-108. Fout = Fin * div_select/2.0.

图 16.1.4.3 寄存器 CCM_ANALOG_PLL_ARMn

在寄存器 CCM_ANALOG_PLL_ARMn 中重要的位如下：

ENABLE: 时钟输出使能位，此位设置为 1 使能 PLL1 输出，如果设置为 0 的话就关闭 PLL1 输出。

DIV_SELECT: 此位设置 PLL1 的输出频率，可设置范围为：54~108，PLL1 CLK = Fin * div_seclec/2.0，Fin=24MHz。如果 PLL1 要输出 1056MHz 的话，div_select 就要设置为 88。

在修改 PLL1 时钟频率的时候我们需要先将内核时钟源改为其他的时钟源，PLL1 可选择的时钟源如图 16.1.4.4 所示：

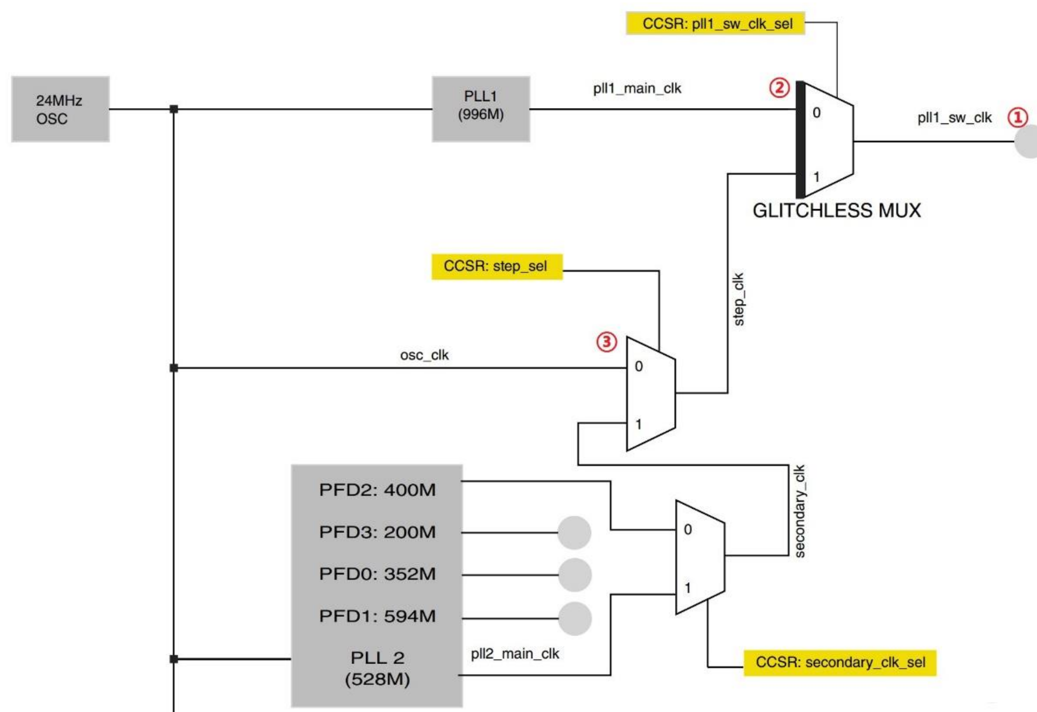


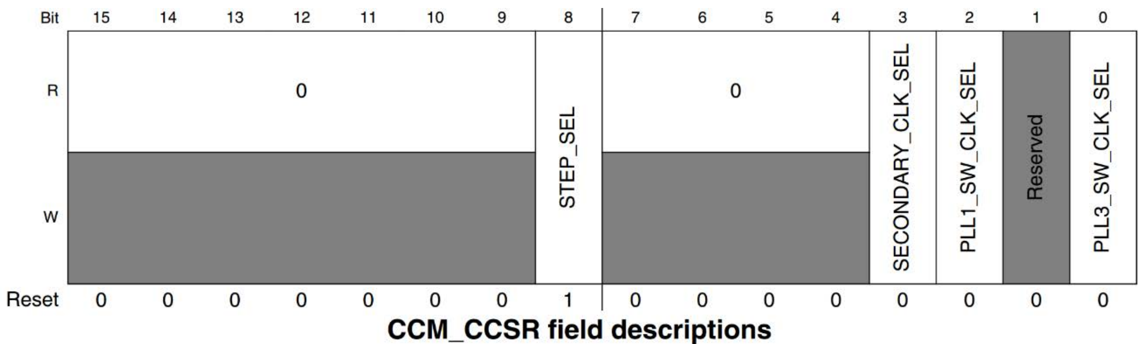
图 16.1.4.4 PLL1 时钟开关

①、pll1_sw_clk 也就是 PLL1 的最终输出频率。

②、此处是一个选择器，选择 pll1_sw_clk 的时钟源，由寄存器 CCM_CCSR 的 PLL1_SW_CLK_SEL 位决定 pll1_sw_clk 是选择 pll1_main_clk 还是 step_clk。正常情况下应该选择 pll1_main_clk，但是如果要对 pll1_main_clk(PLL1)的频率进行调整的话，比如我们要设置 PLL1=1056MHz，此时就要先将 pll1_sw_clk 切换到 step_clk 上。等 pll1_main_clk 调整完成以后再切换回来。

③、此处也是一个选择器，选择 step_clk 的时钟源，由寄存器 CCM_CCSR 的 STEP_SEL 位来决定 step_clk 是选择 osc_clk 还是 secondary_clk。一般选择 osc_clk，也就是 24MHz 的晶振。

这里我们就用到了一个寄存器 CCM_CCSR，此寄存器结构如图 16.1.4.5 所示：



CCM_CCSR field descriptions

Field	Description
31–9 Reserved	This read-only field is reserved and always has the value 0.
8 STEP_SEL	Selects the option to be chosen for the step frequency when shifting ARM frequency. This will control the step_clk. **NOTE:** This mux is allowed to be changed only if its output is not used, i.e. ARM uses the output of pll1, and step_clk is not used. 0 derive clock from osc_clk (24M) - source for lp_apm. 1 derive clock from secondary_clk
7–4 Reserved	This read-only field is reserved and always has the value 0.
3 SECONDARY_CLK_SEL	Select source to generate secondary_clk 0 PLL2 PFD2 (400 M) 1 PLL2 (528 M)
2 PLL1_SW_CLK_SEL	Selects source to generate pll1_sw_clk. 0 pll1_main_clk 1 step_clk
1 -	This field is reserved. Reserved
0 PLL3_SW_CLK_SEL	Selects source to generate pll3_sw_clk. This bit should only be used for testing purposes. 0 pll3_main_clk 1 pll3 bypass clock

图 16.1.4.5 寄存器 CCM_CCSR 结构图

寄存器 CCM_CCSR 我们只用到了 STEP_SEL、PLL1_SW_CLK_SEL 这两个位，一个是用来选择 step_clk 时钟源的，一个是用来选择 pll1_sw_clk 时钟源的。

到这里，修改 I.MX6U 主频的步骤就很清晰了，修改步骤如下：

①、 设置寄存器 CCSR 的 STEP_SEL 位，设置 step_clk 的时钟源为 24M 的晶振。

②、设置寄存器 CCSR 的 PLL1_SW_CLK_SEL 位，设置 pll1_sw_clk 的时钟源为 step_clk=24MHz，通过这一步我们就将 I.MX6U 的主频先设置为 24MHz，直接来自于外部的 24M 晶振。

③、设置寄存器 CCM_ANALOG_PLL_ARMn，将 pll1_main_clk(PLL1)设置为 1056MHz。

④、设置寄存器 CCSR 的 PLL1_SW_CLK_SEL 位，重新将 pll1_sw_clk 的时钟源切换回 pll1_main_clk，切换回来以后的 pll1_sw_clk 就等于 1056MHz。

⑤、最后设置寄存器 CCM_CACRR 的 ARM_PODF 为 2 分频，I.MX6U 的内核主频就为 1056/2=528MHz。

16.1.5 PFD 时钟设置

设置好主频以后我们还需要设置好其他的 PLL 和 PFD 时钟，PLL1 上一小节已经设置了，

PLL2、PLL3 和 PLL7 固定为 528MHz、480MHz 和 480MHz，PLL4~PLL6 都是针对特殊外设的，用到的时候再设置。因此，接下来重点就是设置 PLL2 和 PLL3 的各自 4 路 PFD，NXP 推荐的这 8 路 PFD 频率如表 16.1.5.1 所示：

PFD	NXP 推荐频率值
PLL2_PFD0	352MHz
PLL2_PFD1	594MHz
PLL2_PFD2	400MHz(实际为 396MHz)
PLL2_PFD3	297MHz
PLL3_PFD0	720MHz
PLL3_PFD1	540MHz
PLL3_PFD2	508.2MHz
PLL3_PFD3	454.7MHz

表 16.1.5.1 NXP 推荐的 PFD 频率

先设置 PLL2 的 4 路 PFD 频率，用到寄存器是 CCM_ANALOG_PFD_528n，寄存器结构如图 16.1.5.1 所示：

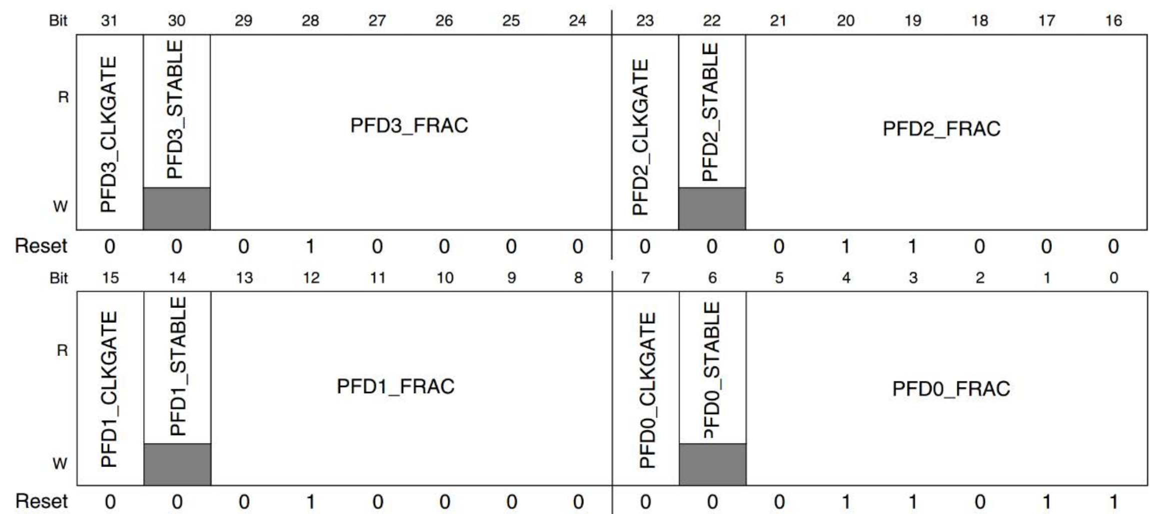


图 16.1.5.1 寄存器 CCM_ANALOG_PFD_528n 结构

从图 16.1.5.1 可以看出，寄存器 CCM_ANALOG_PFD_528n 其实分为四组，分别对应 PFD0~PFD3，每组 8 个 bit，我们就以 PFD0 为例，看一下如何设置 PLL2_PFD0 的频率。PFD0 对应的寄存器位如下：

PFD0_FRAC: PLL2_PFD0 的分频数，PLL2_PFD0 的计算公式为 528*18/PFD0_FRAC，此为可设置的范围为 12~35。如果 PLL2_PFD0 的频率要设置为 352MHz 的话 PFD0_FRAC=528*18/352=27。

PFD0_STABLE: 此位为只读位，可以通过读取此位判断 PLL2_PFD0 是否稳定。

PFD0_CLKGATE: PLL2_PFD0 输出使能位，为 1 的时候关闭 PLL2_PFD0 的输出，为 0 的时候使能输出。

如果我们要设置 PLL2_PFD0 的频率为 352MHz 的话就需要设置 PFD0_FRAC 为 27，PFD0_CLKGATE 为 0。PLL2_PFD1~PLL2_PFD3 设置类似，频率计算公式都是 528*18/PFDX_FRAC(X=1~3)，因此 PLL2_PFD1=594MHz 的话，PFD1_FRAC=16；PLL2_PFD2=400MHz 的话 PFD2_FRAC 不能整除，因此取最近的整数值，即 PFD2_FRAC=24，这样 PLL2_PFD2 实际为 396MHz；PLL2_PFD3=297MHz 的话，PFD3_FRAC=32。

接下来设置 PLL3_PFD0~PLL3_PFD3 这 4 路 PFD 的频率，使用到的寄存器是 CCM_ANALOG_PFD_480n，此寄存器结构如图 16.1.5.2 所示：

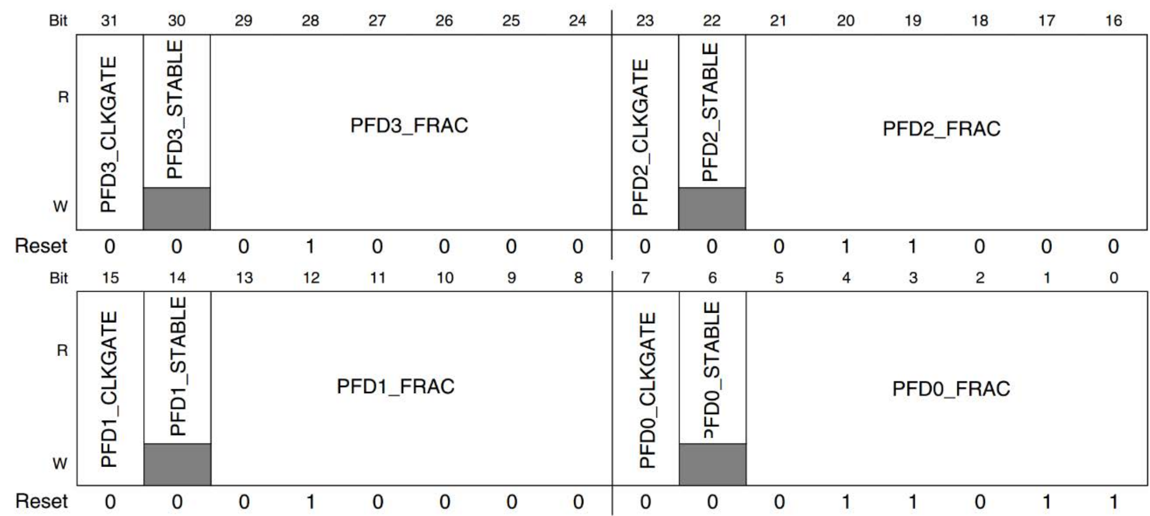


图 16.1.5.2 寄存器 CCM_ANALOG_PFD_480n 结构

从图 16.1.5.2 可以看出，寄存器 CCM_ANALOG_PFD_480n 和 CCM_ANALOG_PFD_528n 的结构是一模一样的，只是一个是 PLL2 的，一个是 PLL3 的。寄存器位的含义也是一样的，只是频率计算公式不同，比如 PLL3_PFDX=480*18/PFDX_FRAC(X=0~3)。如果 PLL3_PFD0=720MHz 的话，PFD0_FRAC=12；如果 PLL3_PFD1=540MHz 的话，PFD1_FRAC=16；如果 PLL3_PFD2=508.2MHz 的话，PFD2_FRAC=17；如果 PLL3_PFD3=454.7MHz 的话，PFD3_FRAC=19。

16.1.6 AHB、IPG 和 PERCLK 根时钟设置

7 路 PLL 和 8 路 PFD 设置完成以后最后还需要设置 AHB_CLK_ROOT 和 IPG_CLK_ROOT 的时钟，I.MX6U 外设根时钟可设置范围如图 16.1.6.1 所示：

Clock Root	Default Frequency (MHz)	Maximum Frequency (MHz)
ARM_CLK_ROOT	12	528
MMDC_CLK_ROOT FABRIC_CLK_ROOT	24	396
AXI_CLK_ROOT	12	264
AHB_CLK_ROOT	6	132
PERCLK_CLK_ROOT	3	66
IPG_CLK_ROOT	3	66
USDHCn_CLK_ROOT	12	198
ACLK_EIM_SLOW_CLK_ROOT	6	132
SPDIF0_CLK_ROOT	1.5	66.6
SAIn_CLK_ROOT	3	66.6
LCDIF_CLK_ROOT	6	150
SIM_CLK_ROOT	12	264
QSPI_CLK_ROOT	12	396
ENFC_CLK_ROOT	12	198
CAN_CLK_ROOT	1.5	80
ECSPI_CLK_ROOT	3	60
UART_CLK_ROOT	4	80

图 16.1.6.1 外设根时钟可设置范围

图 16.1.6.1 给出了大多数外设的根时钟设置范围，AHB_CLK_ROOT 最高可以设置 132MHz，IPG_CLK_ROOT 和 PERCLK_CLK_ROOT 最高可以设置 66MHz。那我们就将 AHB_CLK_ROOT、IPG_CLK_ROOT 和 PERCLK_CLK_ROOT 分别设置为 132MHz、66MHz、66MHz。AHB_CLK_ROOT 和 IPG_CLK_ROOT 的涉及如图 16.1.6.2 所示：

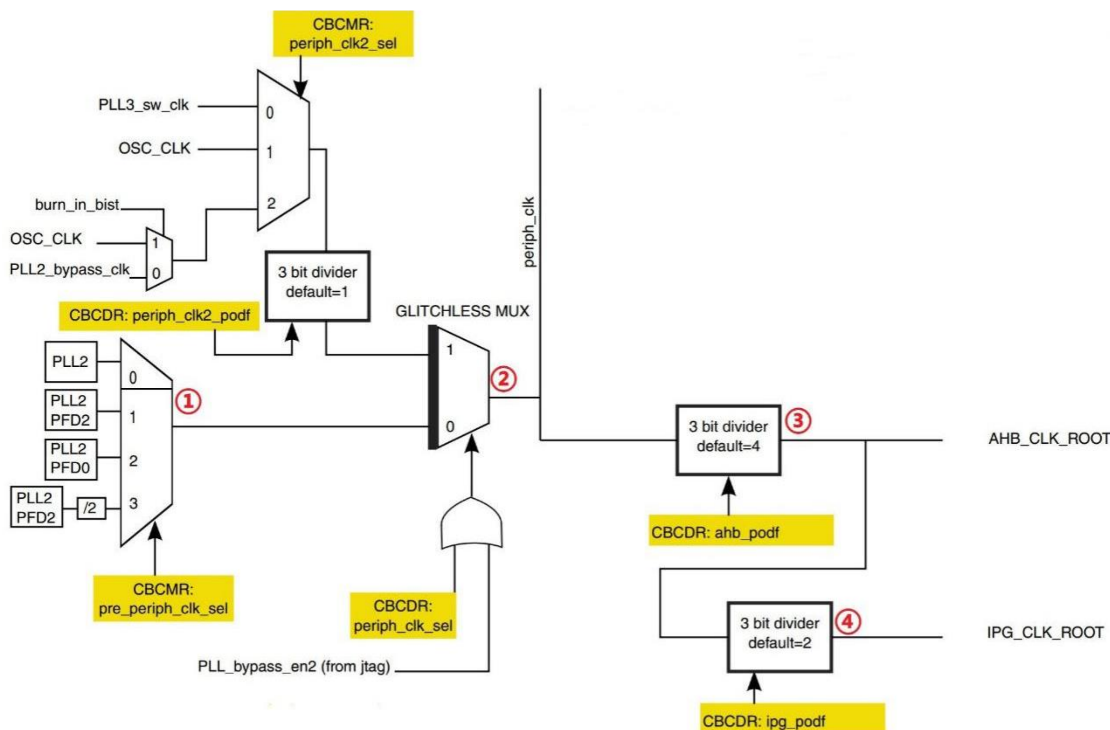


图 16.1.6.2 总线时钟图

图 16.1.6.2 就是 AHB_CLK_ROOT 和 IPG_CLK_ROOT 的时钟图，图中分为了 3 部分。

①、此选择器用来选择 pre_periph_clk 的时钟源，可以选择 PLL2、PLL2_PFD2、PLL2_PFD0 和 PLL2_PFD2/2。寄存器 CCM_CBCMR 的 PRE_PERIPH_CLK_SEL 位决定选择哪一个，默认选择 PLL2_PFD2，因此 pre_periph_clk=PLL2_PFD2=396MHz。

②、此选择器用来选择 periph_clk 的时钟源，由寄存器 CCM_CBCDR 的 PERIPH_CLK_SEL 位与 PLL_bypass_en2 组成的或来选择。当 CCM_CBCDR 的 PERIPH_CLK_SEL 位为 0 的时候 periph_clk=pr_periph_clk=396MHz。

③、通过 CBCDR 的 AHB_PODF 位来设置 AHB_CLK_ROOT 的分频值，可以设置 1~8 分频，如果想要 AHB_CLK_ROOT=132MHz 的话就应该设置为 3 分频：396/3=132MHz。图 16.1.2 中虽然写的是默认 4 分频，但是 I.MX6U 的内部 boot rom 将其改为了 3 分频！

④、通过 CBCDR 的 IPG_PODF 位来设置 IPG_CLK_ROOT 的分频值，可以设置 1~4 分频，IPG_CLK_ROOT 时钟源是 AHB_CLK_ROOT，要想 IPG_CLK_ROOT=66MHz 的话就应该设置 2 分频：132/2=66MHz。

最后要设置的就是 PERCLK_CLK_ROOT 时钟频率，其时钟结构图如图 16.1.6.3 所示：

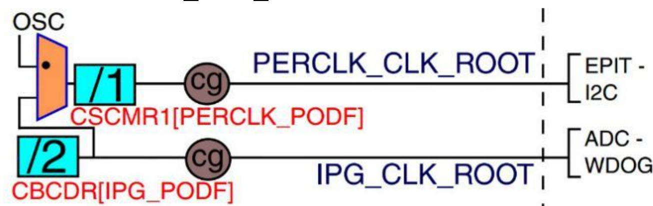


图 16.1.6.3 PERCLK_CLK_ROOT 时钟结构

从图 16.1.6.3 可以看出，PERCLK_CLK_ROOT 来源有两种：OSC(24MHz) 和 IPG_CLK_ROOT，由寄存器 CCM_CSCMR1 的 PERCLK_CLK_SEL 位来决定，如果为 0 的话 PERCLK_CLK_ROOT 的时钟源就是 IPG_CLK_ROOT=66MHz。可以通过寄存器 CCM_CSCMR1 的 PERCLK_PODF 位来设置分频,如果要设置 PERCLK_CLK_ROOT 为 66MHz 的话就要设置为 1 分频。

在上面的设置中用到了三个寄存器：CCM_CBCDR、CCM_CBCMR 和 CCM_CSCMR1，我们依次来看一下这些寄存器，CCM_CBCDR 寄存器结构如图 16.1.6.4 所示：

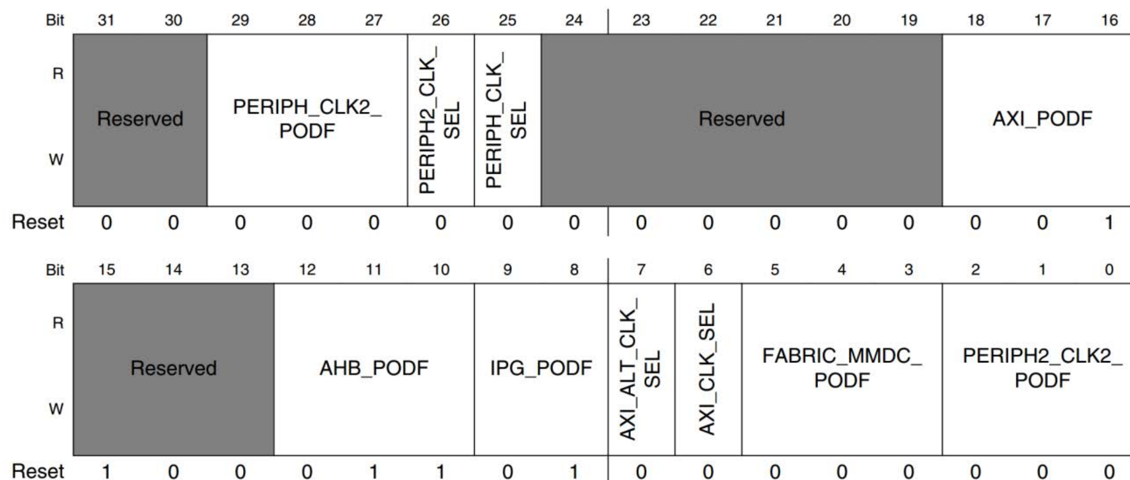


图 16.1.6.4 寄存器 CCM_CBCDR 结构

寄存器 CCM_CBCDR 各个位的含义如下：

PERIPH_CLK2_PODF：periph2 时钟分频，可设置 0~7，分别对应 1~8 分频。

PERIPH2_CLK_SEL：选择 peripheral2 的主时钟，如果为 0 的话选择 PLL2，如果为 1 的话选择 periph2_clk2_clk。修改此位会引起一次与 MMDC 的握手，所以修改完成以后要等待握手完成，握手完成信号由寄存器 CCM_CDHIPR 中指定位表示。

PERIPH_CLK_SEL：peripheral 主时钟选择，如果为 0 的话选择 PLL2，如果为 1 的话选择 periph_clk2_clock。修改此位会引起一次与 MMDC 的握手，所以修改完成以后要等待握手完成，握手完成信号由寄存器 CCM_CDHIPR 中指定位表示。

AXI_PODF：axi 时钟分频，可设置 0~7，分别对应 1~8 分频。

AHB_PODF：ahb 时钟分频，可设置 0~7，分别对应 1~8 分频。修改此位会引起一次与 MMDC 的握手，所以修改完成以后要等待握手完成，握手完成信号由寄存器 CCM_CDHIPR 中指定位表示。

IPG_PODF：ipg 时钟分频，可设置 0~3，分别对应 1~4 分频。

AXI_ALT_CLK_SEL：axi_alt 时钟选择，为 0 的话选择 PLL2_PFD2，如果为 1 的话选择 PLL2_PFD1。

AXI_CLK_SEL：axi 时钟源选择，为 0 的话选择 periph_clk，为 1 的话选择 axi_alt 时钟。

FABRIC_MMDC_PODF：fabric/mmdc 时钟分频设置，可设置 0~7，分别对应 1~8 分频。

PERIPH2_CLK2_PODF：periph2_clk2 的时钟分频，可设置 0~7，分别对应 1~8 分频。

接下来看一下寄存器 CCM_CBCMR，寄存器结构如图 16.1.6.5 所示：

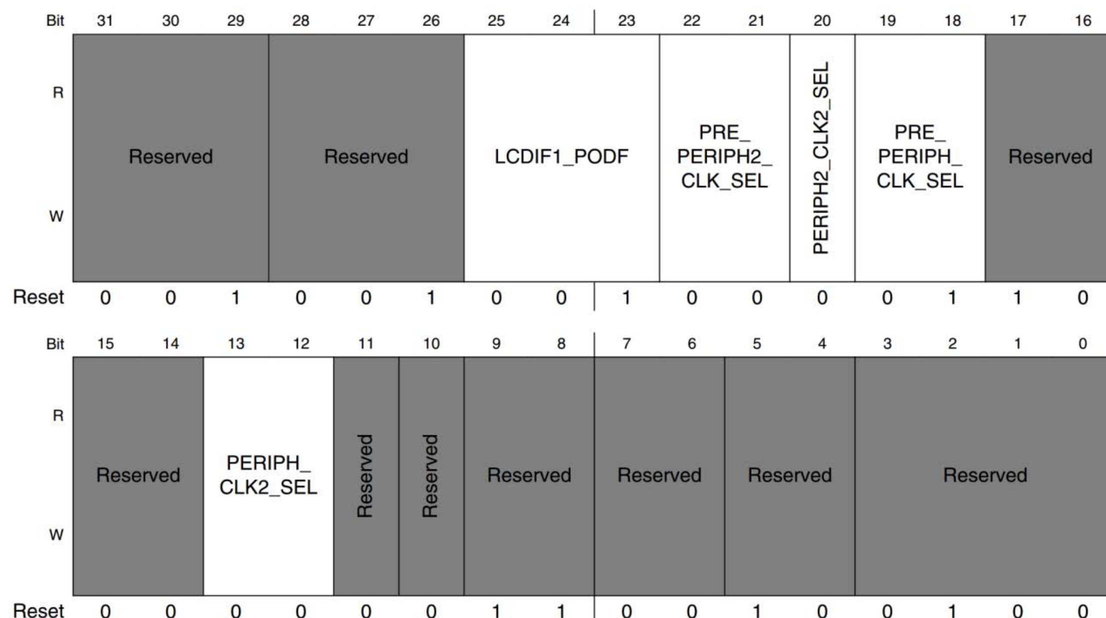


图 16.1.6.5 寄存器 CCM_CBCMR 结构

寄存器 CCM_CBCMR 各个位的含义如下：

LCDIF1_PODF：lcdif1 的时钟分频，可设置 0~7，分别对应 1~8 分频。

PRE_PERIPH2_CLK_SEL：pre_periph2 时钟源选择，00 选择 PLL2，01 选择 PLL2_PFD2，10 选择 PLL2_PFD0，11 选择 PLL4。

PERIPH2_CLK2_SEL：periph2_clk2 时钟源选择为 0 的时候选择 pll3_sw_clk，为 1 的时候选择 OSC。

PRE_PERIPH_CLK_SEL：pre_periph 时钟源选择，00 选择 PLL2，01 选择 PLL2_PFD2，10 选择 PLL2_PFD0，11 选择 PLL2_PFD2/2。

PERIPH_CLK2_SEL：peripheral_clk2 时钟源选择，00 选择 pll3_sw_clk，01 选择 osc_clk，10 选择 pll2_bypass_clk。

最后看一下寄存器 CCM_CSCMR1，寄存器结构如图 16.1.6.6 所示：

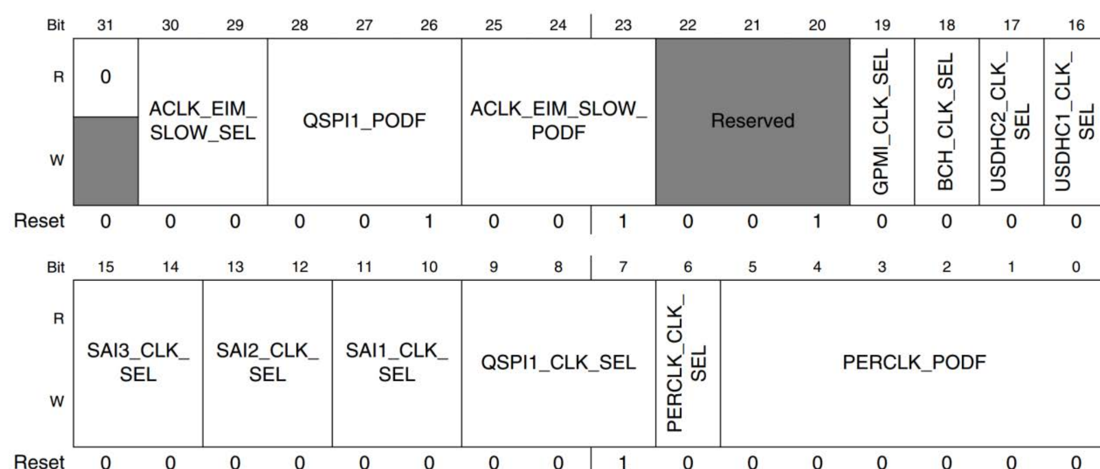


图 16.1.6.6 寄存器 CCM_CSCMR1 结构

此寄存器主要用于外设时钟源的选择，比如 QSPI1、ACLK、GPMI、BCH 等外设，我们重点看一下下面另个未：

PERCLK_CK_SEL：perclk 时钟源选择，为 0 的话选择 ipg clk，为 1 的话选择 osc clk。

PERCLK_PODF：perclk 的时钟分频，可设置 0~7，分别对应 1~8 分频。

在修改如下时钟选择器或者分频器的时候会引起与 MMDC 的握手发生：

①、mmdc_podf

②、periph_clk_sel

③、periph2_clk_sel

④、arm_podf

⑤、ahb_podf

发生握手信号以后需要等待握手完成，寄存器 CCM_CDHIPR 中保存着握手信号是否完成，如果相应的位为 1 的话就表示握手没有完成，如果为 0 的话就表示握手完成，很简单，这里就不详细的列举寄存器 CCM_CDHIPR 中的各个位了。

另外在修改 arm_podf 和 ahb_podf 的时候需要先关闭其时钟输出，等修改完成以后再打开，否则的话可能会出现在修改完成以后没有时钟输出的问题。本教程需要修改寄存器 CCM_CBCDR 的 AHB_PODF 位来设置 AHB_ROOT_CLK 的时钟，所以在修改之前必须先关闭 AHB_ROOT_CLK 的输出。但是笔者没有找到相应的寄存器，因此目前没法关闭，那也就没法设置 AHB_PODF 了。不过 AHB_PODF 内部 boot rom 设置为了 3 分频，如果 pre_periph_clk 的时钟源选择 PLL2_PFD2 的话，AHB_ROOT_CLK 也是 396MHz/3=132MHz。

至此，I.MX6U 的时钟系统就讲解完了，I.MX6U 的时钟系统还是很复杂的，大家要结合《I.MX6ULL 参考手册》中时钟相关的结构图来学习。本章我们也只是讲解了如何进行主频、PLL、PFD 和一些总线时钟的设置，关于具体的外设时钟设置我们在学习到的时候在详细的讲解。

16.2 硬件原理分析

时钟原理图分析参考 16.1.1 小节。

16.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 8_clk。

本试验在上一章试验“7_key”的基础上完成，因为本试验是配置 I.MX6U 的系统时钟，因此我们直接在文件“bsp_clk.c”上做修改，修改 bsp_clk.c 的内容如下：

示例代码 16.3.1 bsp_clk.c 文件代码

```
1  #include "bsp_clk.h"
2  
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名   : bsp_clk.c
6  作者     : 左忠凯
7  版本     : V1.0
8  描述     : 系统时钟驱动。
9  其他     : 无
10 论坛     : www.openedv.com
11 日志     : 初版 V1.0 2019/1/3 左忠凯创建
12 
13            V2.0    2019/1/3 左忠凯修改
14            添加了函数 imx6u_clkinit()，完成 I.MX6U 的系统时钟初始化
```

```
15 ****************************************************************/
16 
17 /*
18  * @description : 使能 I.MX6U 所有外设时钟
19  * @param       : 无
20  * @return      : 无
21  */
22 void clk_enable(void)
23 {
24     CCM->CCGR0 = 0XFFFFFFFF;
25     CCM->CCGR1 = 0XFFFFFFFF;
26     CCM->CCGR2 = 0XFFFFFFFF;
27     CCM->CCGR3 = 0XFFFFFFFF;
28     CCM->CCGR4 = 0XFFFFFFFF;
29     CCM->CCGR5 = 0XFFFFFFFF;
30     CCM->CCGR6 = 0XFFFFFFFF;
31 }
32 
33 /*
34  * @description : 初始化系统时钟 528Mhz，并且设置 PLL2 和 PLL3 各个
35                   PFD 时钟,所有的时钟频率均按照 I.MX6U 官方手册推荐的值.
36  * @param       : 无
37  * @return      : 无
38  */
39 void imx6u_clkinit(void)
40 {
41     unsigned int reg = 0;
42     /* 1、设置 ARM 内核时钟为 528MHz */
43     /* 1.1、判断当使用哪个时钟源启动的，正常情况下是由 pll1_sw_clk 驱动的，而
44      *      pll1_sw_clk 有两个来源：pll1_main_clk 和 tep_clk，如果要
45      *      让内核跑到 528M，那必须选择 pll1_main_clk 作为 pll1 的时钟源。
46      *      如果我们要修改 pll1_main_clk 时钟的话就必须先将 pll1_sw_clk 从
47      *      pll1_main_clk 切换到 step_clk,当修改完以后再将 pll1_sw_clk 切换
48      *      回 pll1_main_cl，step_clk 等于 24MHz。
49      */
50 
51     if((((CCM->CCSR) >> 2) & 0x1 ) == 0)   /*   pll1_main_clk? */
52     {
53         CCM->CCSR &= ~(1 << 8);  /* 配置 step_clk 时钟源为 24MH OSC */
54         CCM->CCSR |= (1 << 2);   /* 配置 pll1_sw_clk 时钟源为 step_clk */
55     }
56 
57     /* 1.2、设置 pll1_main_clk 为 1056MHz,也就是 528*2=1056MHZ,
```

```
58        *      因为 pll1_sw_clk 进 ARM 内核的时候会被二分频！
59        *      配置 CCM_ANLOG->PLL_ARM 寄存器
60        *      bit13: 1 使能时钟输出
61        *      bit[6:0]: 88, 由公式：Fout = Fin * div_select / 2.0，
62        *      1056=24*div_select/2.0, 得出：div_select=88。
63        */
64       CCM_ANALOG->PLL_ARM = (1 << 13) | ((88 << 0) & 0X7F);
65       CCM->CCSR &= ~(1 << 2);/* 将 pll_sw_clk 时钟切换回 pll1_main_clk */
66       CCM->CACRR = 1;    /* ARM 内核时钟为 pll1_sw_clk/2=1056/2=528Mhz */
67
68       /* 2、设置 PLL2(SYS PLL)各个 PFD */
69       reg = CCM_ANALOG->PFD_528;
70       reg &= ~(0X3F3F3F3F);    /* 清除原来的设置                */
71       reg |= 32<<24;           /* PLL2_PFD3=528*18/32=297Mhz   */
72       reg |= 24<<16;           /* PLL2_PFD2=528*18/24=396Mhz   */
73       reg |= 16<<8;            /* PLL2_PFD1=528*18/16=594Mhz   */
74       reg |= 27<<0;            /* PLL2_PFD0=528*18/27=352Mhz   */
75       CCM_ANALOG->PFD_528=reg; /* 设置 PLL2_PFD0~3             */
76
77       /* 3、设置 PLL3(USB1)各个 PFD */
78       reg = 0;                  /* 清零   */
79       reg = CCM_ANALOG->PFD_480;
80       reg &= ~(0X3F3F3F3F);    /* 清除原来的设置                    */
81       reg |= 19<<24;           /* PLL3_PFD3=480*18/19=454.74Mhz    */
82       reg |= 17<<16;           /* PLL3_PFD2=480*18/17=508.24Mhz    */
83       reg |= 16<<8;            /* PLL3_PFD1=480*18/16=540Mhz       */
84       reg |= 12<<0;            /* PLL3_PFD0=480*18/12=720Mhz       */
85       CCM_ANALOG->PFD_480=reg;    /* 设置 PLL3_PFD0~3              */
86
87       /* 4、设置 AHB 时钟 最小 6Mhz， 最大 132Mhz */
88       CCM->CBCMR &= ~(3 << 18); /* 清除设置*/
89       CCM->CBCMR |= (1 << 18);   /* pre_periph_clk=PLL2_PFD2=396MHz */
90       CCM->CBCDR &= ~(1 << 25); /* periph_clk=pre_periph_clk=396MHz */
91       while(CCM->CDHIPR & (1 << 5));/* 等待握手完成 */
92
93       /* 修改 AHB_PODF 位的时候需要先禁止 AHB_CLK_ROOT 的输出，但是
94        * 我没有找到关闭 AHB_CLK_ROOT 输出的的寄存器，所以就没法设置。
95        * 下面设置 AHB_PODF 的代码仅供学习参考不能直接拿来使用！！
96        * 内部 boot rom 将 AHB_PODF 设置为了 3 分频，即使我们不设置 AHB_PODF，
97        * AHB_ROOT_CLK 也依旧等于 396/3=132Mhz。
98        */
99   #if 0
100      /* 要先关闭 AHB_ROOT_CLK 输出，否则时钟设置会出错 */
```

```
101     CCM->CBCDR &= ~(7 << 10);/* CBCDR 的 AHB_PODF 清零 */
102     CCM->CBCDR |= 2 << 10; /* AHB_PODF 3 分频，AHB_CLK_ROOT=132MHz */
103     while(CCM->CDHIPR & (1 << 1));/* 等待握手完成 */
104 #endif
105 
106     /* 5、设置 IPG_CLK_ROOT 最小 3Mhz，最大 66Mhz */
107     CCM->CBCDR &= ~(3 << 8);/* CBCDR 的 IPG_PODF 清零 */
108     CCM->CBCDR |= 1 << 8;    /* IPG_PODF 2 分频，IPG_CLK_ROOT=66MHz */
109 
110     /* 6、设置 PERCLK_CLK_ROOT 时钟 */
111     CCM->CSCMR1 &= ~(1 << 6);   /* PERCLK_CLK_ROOT 时钟源为 IPG */
112     CCM->CSCMR1 &= ~(7 << 0);   /* PERCLK_PODF 位清零，即 1 分频 */
113 }
```

文件 bsp_clk.c 中一共有两个函数：clk_enable 和 imx6u_clkinit，其中函数 clk_enable 前面已经讲过了，就是使能 I.MX6U 的所有外设时钟。函数 imx6u_clkinit 才是本章的重点，imx6u_clkinit 先设置系统主频为 528MHz，然后根据我们上一小节分析的 I.MX6U 时钟系统来设置 8 路 PFD，最后设置 AHB、IPG 和 PERCLK 的时钟频率。

在 bsp_clk.h 文件中添加函数 imx6u_clkinit 的声明，最后修改 main.c 文件，在 main 函数里面调用 imx6u_clkinit 来初始化时钟，如下所示：

示例代码 16.3.2 main 函数

```
1  int main(void)
2  {
3      int i = 0;
4      int keyvalue = 0;
5      unsigned char led_state = OFF;
6      unsigned char beep_state = OFF;
7  
8      imx6u_clkinit();       /* 初始化系统时钟        */
9      clk_enable();          /* 使能所有的时钟        */
10     led_init();            /* 初始化 led            */
11     beep_init();           /* 初始化 beep           */
12     key_init();            /* 初始化 key            */
13 
14     /* 省略掉其它代码 */
15 }
```

上述代码的第 8 行就是时钟初始化函数，时钟初始化函数最好放到最开始的地方调用。

16.4 编译下载验证

16.4.1 编写 Makefile 和链接脚本

因为本章是在试验“7_key”上修改的，而且本章试验没有添加任何新的文件，因此只需要修改 Makefile 的变量 TARGET 为“clk”即可，如下所示：

```
TARGET          ?= clk
```

链接脚本保持不变。

16.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 clk.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload clk.bin /dev/sdd       //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。本试验效果其实和试验“7_key”一样，但是 LED 灯的闪烁频率相比试验“7_key”要快一点。因为试验“7_key”的主频是 396MHz，而本试验的主频被配置成了 528MHz，因此代码执行速度会变快，所以延时函数的运行就会加快。

第十七章 GPIO 中断试验

中断系统是一个处理器重要的组成部分，中断系统极大的提高了 CPU 的执行效率，在学习 STM32 的时候就经常用到中断。本章就通过与 STM32 的对比来学习一下 Cortex-A7(I.MX6U) 中断系统和 Cortex-M(STM32)中断系统的异同，同时，本章会将 I.MX6U 的一个 IO 作为输入中断，借此来讲解如何对 I.MX6U 的中断系统进行编程。

17.1 Cortex-A7 中断系统详解

17.1.1 STM32 中断系统回顾

STM32 的中断系统主要有一下几个关键点：

、中断向量表。

、NVIC(内嵌向量中断控制器)。

、中断使能。

、中断服务函数。

1、中断向量表

中断向量表是一个表，这个表里面存放的是中断向量。中断服务程序的入口地址或存放中断服务程序的首地址成为中断向量，因此中断向量表是一系列中断服务程序入口地址组成的表。这些中断服务程序(函数)在中断向量表中的位置是由半导体厂商定好的，当某个中断被触发以后就会自动跳转到中断向量表中对应的中断服务程序(函数)入口地址处。中断向量表在整个程序的最前面，比如 STM32F103 的中断向量表如下所示：

示例代码 17.1.1.1 STM32F103 中断向量表

```
1  __Vectors   DCD    __initial_sp         ; Top of Stack
2              DCD    Reset_Handler        ; Reset Handler
3              DCD    NMI_Handler          ; NMI Handler
4              DCD    HardFault_Handler    ; Hard Fault Handler
5              DCD    MemManage_Handler    ; MPU Fault Handler
6              DCD    BusFault_Handler     ; Bus Fault Handler
7              DCD    UsageFault_Handler   ; Usage Fault Handler
8              DCD    0                    ; Reserved
9              DCD    0                    ; Reserved
10             DCD    0                    ; Reserved
11             DCD    0                    ; Reserved
12             DCD    SVC_Handler          ; SVCall Handler
13             DCD    DebugMon_Handler     ; Debug Monitor Handler
14             DCD    0                    ; Reserved
15             DCD    PendSV_Handler       ; PendSV Handler
16             DCD    SysTick_Handler      ; SysTick Handler
17
18             ; External Interrupts
19             DCD    WWDG_IRQHandler      ; Window Watchdog
20             DCD    PVD_IRQHandler     ; PVD through EXTI Line detect
21             DCD    TAMPER_IRQHandler    ; Tamper
22             DCD    RTC_IRQHandler       ; RTC
23             DCD    FLASH_IRQHandler     ; Flash
24
25             /* 省略掉其它代码 */
26
27             DCD    DMA2_Channel4_5_IRQHandler ; DMA2 Channel4 & l5
```

```
28 __Vectors_End
```

“示例代码 17.1.1.1”就是 STM32F103 的中断向量表，中断向量表都是链接到代码的最前面，比如一般 ARM 处理器都是从地址 0X00000000 开始执行指令的，那么中断向量表就是从 0X00000000 开始存放的。“示例代码 17.1.1.1”中第 1 行的“__initial_sp”就是第一条中断向量，存放的是栈顶指针，接下来是第 2 行复位中断复位函数 Reset_Handler 的入口地址，依次类推，直到第 27 行的最后一个中断服务函数 DMA2_Channel4_5_IRQHandler 的入口地址，这样 STM32F103 的中断向量表就建好了。

我们说 ARM 处理器都是从地址 0X00000000 开始运行的，但是我们学习 STM32 的时候代码是下载到 0X8000000 开始的存储区域中。因此中断向量表是存放到 0X8000000 地址处的，而不是 0X00000000，这样不是就出错了吗？为了解决这个问题，Cortex-M 架构引入了一个新的概念——中断向量表偏移，通过中断向量表偏移就可以将中断向量表存放到任意地址处，中断向量表偏移配置在函数 SystemInit 中完成，通过向 SCB_VTOR 寄存器写入新的中断向量表首地址即可，代码如下所示：

示例代码 17.1.1.2 STM32F103 中断向量表偏移

```
1  void SystemInit (void)
2  {
3      RCC->CR |= (uint32_t)0x00000001;
4
5      /* 省略其它代码 */
6
7  #ifdef VECT_TAB_SRAM
8      SCB->VTOR = SRAM_BASE | VECT_TAB_OFFSET;
9  #else
10     SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
11 #endif
12 }
```

第 8 行和第 10 行就是设置中断向量表偏移，第 8 行是将中断向量表设置到 RAM 中，第 10 行是将中断向量表设置到 ROM 中，基本都是将中断向量表设置到 ROM 中，也就是地址 0X8000000 处。第 10 行用到了 FALSH_BASE 和 VECT_TAB_OFFSET，这两个都是宏，定义如下所示：

```
#define FLASH_BASE          ((uint32_t)0x08000000)
#define VECT_TAB_OFFSET     0x0
```

因此第 10 行的代码就是：SCB->VTOR=0X080000000，中断向量表偏移设置完成。通过上面的讲解我们了解了两个跟 STM32 中断有关的概念：中断向量表和中断向量表偏移，那么这个跟 I.MX6U 有什么关系呢？因为 I.MX6U 所使用的 Cortex-A7 内核也有中断向量表和中断向量表偏移，而且其含义和 STM32 是一模一样的！只是用到的寄存器不通而已，概念完全相同！

2、NVIC(内嵌向量中断控制器)

中断系统得有个管理机构，对于 STM32 这种 Cortex-M 内核的单片机来说这个管理机构叫做 NVIC，全称叫做 Nested Vectored Interrupt Controller。关于 NVIC 本教程不作详细的讲解，既然 Cortex-M 内核有个中断系统的管理机构—NVIC，那么 I.MX6U 所使用的 Cortex-A7 内核是不是也有个中断系统管理机构？答案是肯定的，不过 Cortex-A 内核的中断管理机构不叫做 NVIC，而是叫做 GIC，全称是 general interrupt controller，后面我们会详细的讲解 Cortex-A 内核的 GIC。

3、中断使能

要使用某个外设的中断，肯定要先使能这个外设的中断，以 STM32F103 的 PE2 这个 IO 为例，假如我们要使用 PE2 的输入中断肯定要使用如下代码来使能对应的中断：

```
NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;  //抢占优先级 2，
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;          //子优先级 2
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                //使能外部中断通道
NVIC_Init(&NVIC_InitStructure);
```

上述代码就是使能 PE2 对饮过的 EXTI2 中断，同理，如果要使用 I.MX6U 的某个中断的话也需要使能其对应的中断。

4、中断服务函数

我们使用中断的目的就是为了使用中断服务函数，当中断发生以后中断服务函数就会被调用，我们要处理的工作就可以放到中断服务函数中去完成。同样以 STM32F103 的 PE2 为例，其中断服务函数如下所示：

```
/* 外部中断 2 服务程序 */
void EXTI2_IRQHandler(void)
{
    /* 中断处理代码 */
}
```

当 PE2 引脚的中断触发以后就会调用其对应的中断处理函数 EXTI2_IRQHandler，我们可以在函数 EXTI2_IRQHandler 中添加中断处理代码。同理，I.MX6U 也有中断服务函数，当某个外设中断发生以后就会调用其对应的中断服务函数。

通过对 STM32 中断系统的回顾，我们知道了 Cortex-M 内核的中断处理过程，那么 Cortex-A 内核的中断处理过程是否是一样的，有什么异同呢？接下来我们带着这样的疑问来学习 Cortex-A7 内核的中断系统。

17.1.2 Cortex-A7 中断系统简介

跟 STM32 一样，Cortex-A7 也有中断向量表，中断向量表也是在代码的最前面。Cortex-A7 内核有 8 个异常中断，这 8 个异常中断的中断向量表如表 17.1.2.1 所示：

向量地址	中断类型	中断模式
0X00	复位中断(Rest)	特权模式(SVC)
0X04	未定义指令中断(Undefined Instruction)	未定义指令中止模式(Undef)
0X08	软中断(Software Interrupt,SWI)	特权模式(SVC)
0X0C	指令预取中止中断(Prefetch Abort)	中止模式
0X10	数据访问中止中断(Data Abort)	中止模式
0X14	未使用(Not Used)	未使用
0X18	IRQ 中断(IRQ Interrupt)	外部中断模式(IRQ)
0X1C	FIQ 中断(FIQ Interrupt)	快速中断模式(FIQ)

表 17.1.2.1 Cortex-A7 中断向量表

中断向量表里面都是中断服务函数的入口地址，因此一款芯片有什么中断都是可以从中断向量表看出来的。从表 17.1.2.1 中可以看出，Cortex-A7 一共有 8 个中断，而且还有一个中断向

量未使用，实际只有 7 个中断。和“示例代码 17.1.1.1”中的 STM32F103 中断向量表比起来少了很多！难道一个能跑 Linux 的芯片只有这 7 个中断？明显不可能的！那类似 STM32 中的 EXTI9_5_IRQHandler、TIM2_IRQHandler 这样的中断向量在哪里？I2C、SPI、定时器等等的中断怎么处理呢？这个就是 Cortex-A 和 Cotex-M 在中断向量表这一块的区别，对于 Cortex-M 内核来说，中断向量表列举出了一款芯片所有的中断向量，包括芯片外设的所有中断。对于 Cotex-A 内核来说并没有这么做，在表 17.1.2.1 中有个 IRQ 中断， Cortex-A 内核 CPU 的所有外部中断都属于这个 IQR 中断，当任意一个外部中断发生的时候都会触发 IRQ 中断。在 IRQ 中断服务函数里面就可以读取指定的寄存器来判断发生的具体是什么中断，进而根据具体的中断做出相应的处理。这些外部中断和 IQR 中断的关系如图 17.1.2.1 所示：

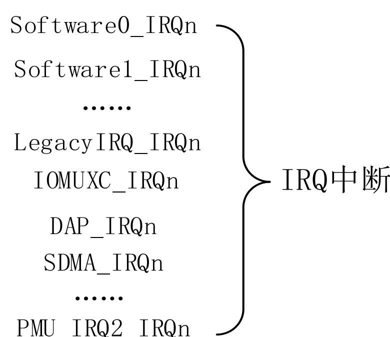


图 17.1.2.1 外部中断和 IRQ 中断关系

在图 17.1.2.1 中，左侧的 Software0_IRQn~PMU_IRQ2_IRQ 这些都是 I.MX6U 的中断，他们都属于 IRQ 中断。当图 17.1.2.1 左侧这些中断中任意一个发生的时候 IRQ 中断都会被触发，所以我们需要在 IRQ 中断服务函数中判断究竟是左侧的哪个中断发生了，然后再做出具体的处理。

在表 17.1.2.1 中一共有 7 个中断，简单介绍一下这 7 个中断：

、复位中断(Rest)，CPU 复位以后就会进入复位中断，我们可以在复位中断服务函数里面做一些初始化工作，比如初始化 SP 指针、DDR 等等。

、未定义指令中断(Undefined Instruction)，如果指令不能识别的话就会产生此中断。

、软中断(Software Interrupt,SWI)，由 SWI 指令引起的中断，Linux 的系统调用会用 SWI 指令来引起软中断，通过软中断来陷入到内核空间。

、指令预取中止中断(Prefetch Abort)，预取指令的出错的时候会产生此中断。

、数据访问中止中断(Data Abort)，访问数据出错的时候会产生此中断。

、IRQ 中断(IRQ Interrupt)，外部中断，前面已经说了，芯片内部的外设中断都会引起此中断的发生。

、FIQ 中断(FIQ Interrupt)，快速中断，如果需要快速处理中断的话就可以使用此中。

在上面的 7 个中断中，我们常用的就是复位中断和 IRQ 中断，所以我们需要编写这两个中断的中断服务函数，稍后我们会讲解如何编写对应的中断服务函数。首先我们要根据表 17.1.2.1 的内容来创建中断向量表，中断向量表处于程序最开始的地方，比如我们前面例程的 start.S 文件最前面，中断向量表如下：

示例代码 17.1.1.1 Cortex-A 向量表模板

```
1  .global _start                /* 全局标号             */
2
3  _start:
4      ldr pc, =Reset_Handler    /* 复位中断             */
```

```
5       ldr pc, =Undefined_Handler  /* 未定义指令中断       */
6       ldr pc, =SVC_Handler        /* SVC(Supervisor)中断   */
7       ldr pc, =PrefAbort_Handler  /* 预取终止中断          */
8       ldr pc, =DataAbort_Handler  /* 数据终止中断          */
9       ldr pc, =NotUsed_Handler    /* 未使用中断            */
10      ldr pc, =IRQ_Handler        /* IRQ 中断              */
11      ldr pc, =FIQ_Handler        /* FIQ(快速中断)未定义中断  */
12
13 /* 复位中断 */
14 Reset_Handler:
15      /*  复位中断具体处理过程 */
16
17 /* 未定义中断 */
18 Undefined_Handler:
19      ldr r0, =Undefined_Handler
20      bx r0
21
22 /* SVC 中断 */
23 SVC_Handler:
24      ldr r0, =SVC_Handler
25      bx r0
26
27 /* 预取终止中断 */
28 PrefAbort_Handler:
29      ldr r0, =PrefAbort_Handler
30      bx r0
31
32 /* 数据终止中断 */
33 DataAbort_Handler:
34      ldr r0, =DataAbort_Handler
35      bx r0
36
37 /* 未使用的中断 */
38 NotUsed_Handler:
39
40      ldr r0, =NotUsed_Handler
41      bx r0
42
43 /* IRQ 中断！重点！！！！！ */
44 IRQ_Handler:
45      /*  复位中断具体处理过程 */
46
47 /* FIQ 中断 */
```

```
48 FIQ_Handler:
49     ldr r0, =FIQ_Handler
50     bx r0
```

第 4 到 11 行是中断向量表，当指定的中断发生以后就会调用对应的中断复位函数，比如复位中断发生以后就会执行第 4 行代码，也就是调用函数 Reset_Handler，函数 Reset_Handler 就是复位中断的中断复位函数，其它的中断同理。

第 14 到 50 行就是对应的中断服务函数，中断服务函数都是用汇编编写的，我们实际需要编写的只有复位中断服务函数 Reset_Handler 和 IRQ 中断服务函数 IRQ_Handler，其它的中断本教程没有用到，所以都是死循环。在编写复位中断复位函数和 IRQ 中断服务函数之前我们还需要了解一些其它的知识，否则的话就没法编写。

17.1.3 GIC 控制器简介

1、GIC 控制器总览

STM32(Cortex-M)的中断控制器叫做 NVIC，I.MX6U(Cortex-A)的中断控制器叫做 GIC，关于 GIC 的详细内容请参考开发板光盘中的文档《ARM Generic Interrupt Controller(ARM GIC 控制器)V2.0.pdf》。

GIC 是 ARM 公司给 Cortex-A/R 内核提供的一个中断控制器，类似 Cortex-M 内核中的 NVIC。目前 GIC 有 4 个版本:V1~V4，V1 是最老的版本，已经被废弃了。V2~V4 目前正在大量的使用。GIC V2 是给 ARMv7-A 架构使用的，比如 Cortex-A7、Cortex-A9、Cortex-A15 等，V3 和 V4 是给 ARMv8-A/R 架构使用的，也就是 64 位芯片使用的。I.MX6U 是 Cortex-A 内核的，因此我们主要讲解 GIC V2。GIC V2 最多支持 8 个核。ARM 会根据 GIC 版本的不同研发出不同的 IP 核，那些半导体厂商直接购买对应的 IP 核即可，比如 ARM 针对 GIC V2 就开发出了 GIC400 这个中断控制器 IP 核。当 GIC 接收到外部中断信号以后就会报给 ARM 内核，但是 ARM 内核只提供了四个信号给 GIC 来汇报中断情况：VFIQ、VIRQ、FIQ 和 IRQ，他们之间的关系如图 17.1.3.1 所示：

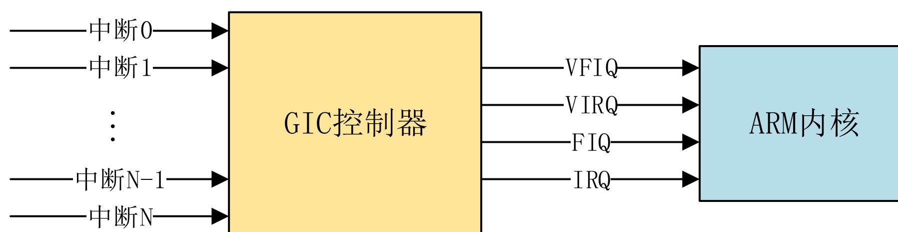


图 17.1.3.1 中断示意图

在图 17.1.3.1 中，GIC 接收众多的外部中断，然后对其进行处理，最终就只通过四个信号报给 ARM 内核，这四个信号的含义如下：

VFIQ:虚拟快速 FIQ。

VIRQ:虚拟快速 IRQ。

FIQ:快速中断 IRQ。

IRQ:外部中断 IRQ。

VFIQ 和 VIRQ 是针对虚拟化的，我们讨论虚拟化，剩下的就是 FIQ 和 IRQ 了，我们前面都讲了很多次了。本教程我们只使用 IRQ，所以相当于 GIC 最终向 ARM 内核就上报一个 IRQ 信号。那么 GIC 是如何完成这个工作的呢？GICV2 的逻辑图如图 17.1.3.2 所示：

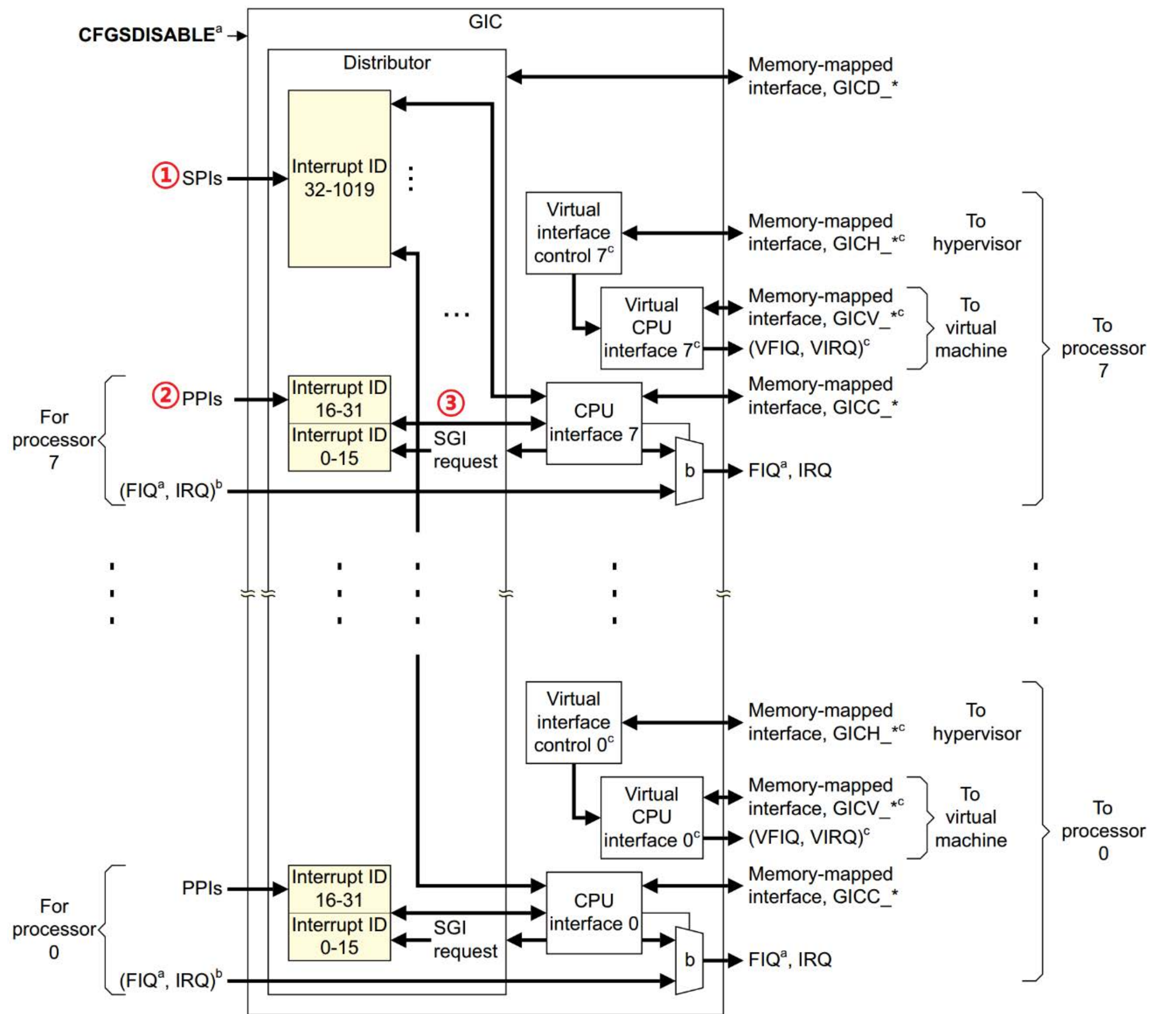


图 17.1.3.2 GICV2 总体框图

图 17.1.3.1 中左侧部分就是中断源，中间部分就是 GIC 控制器，最右侧就是中断控制器向处理器内核发送中断信息。我们重点要看的肯定是中间的 GIC 部分，GIC 将众多的中断源分为分为三类：

①、SPI(Shared Peripheral Interrupt),共享中断，顾名思义，所有 Core 共享的中断，这个是最常见的，那些外部中断都属于 SPI 中断(注意！不是 SPI 总线那个中断) 。比如按键中断、串口中断等等，这些中断所有的 Core 都可以处理，不限定特定 Core。

②、PPI(Private Peripheral Interrupt)，私有中断，我们说了 GIC 是支持多核的，每个核肯定有自己独有的中断。这些独有的中断肯定是要指定的核心处理，因此这些中断就叫做私有中断。

③、SGI(Software-generated Interrupt)，软件中断，由软件触发引起的中断，通过向寄存器 GICD_SGIR 写入数据来触发，系统会使用 SGI 中断来完成多核之间的通信。

2、中断 ID

中断源有很多，为了区分这些不同的中断源肯定要给他们分配一个唯一 ID，这些 ID 就是中断 ID。每一个 CPU 最多支持 1020 个中断 ID，中断 ID 号为 ID0~ID1019。这 1020 个 ID 包含了 PPI、SPI 和 SGI，那么这三类中断是如何分配这 1020 个中断 ID 的呢？这 1020 个 ID 分配如下：

ID0~ID15：这 16 个 ID 分配给 SGI。

ID16~ID31：这 16 个 ID 分配给 PPI。

ID32~ID1019：这 988 个 ID 分配给 SPI，像 GPIO 中断、串口中断等这些外部中断 ，至于具体到某个 ID 对应哪个中断那就由半导体厂商根据实际情况去定义了。比如 I.MX6U 的总共使用了 128 个中断 ID，加上前面属于 PPI 和 SGI 的 32 个 ID，I.MX6U 的中断源共有 128+32=160 个，这 128 个中断 ID 对应的中断在《I.MX6ULL 参考手册》的“3.2 Cortex A7 interrupts”小节，中断源如表 17.1.3.1 所示：

IRQ	ID	中断源	描述
0	32	boot	用于在启动异常的时候通知内核。
1	33	ca7_platform	DAP 中断，调试端口访问请求中断。
2	34	sdma	SDMA 中断。
3	35	tsc	TSC(触摸)中断。
4		snvs_lp_wrapper snvs_hp_wrapper	SNVS 中断。
……		……	……
124	156	无	保留
125	157	无	保留
126	158	无	保留
127	159	pmu	PMU 中断

表 17.1.3.1 I.MX6U 中断源

限于篇幅原因，表 17.1.3.1 中并没有给出 I.MX6U 完整的中断源，完整的中断源自行查阅《I.MX6ULL 参考手册》的 3.2 小节。打开裸机例程“9_int”，我们前面移植了 NXP 官方 SDK 中的文件 MCIMX6Y2C.h，在此文件中定义了一个枚举类型 IRQn_Type，此枚举类型就枚举出了 I.MX6U 的所有中断，代码如下所示：

示例代码 17.1.3.1 中断向量

```
1 #define NUMBER_OF_INT_VECTORS 160    /* 中断源 160 个，SGI+PPI+SPI*/
2
3 typedef enum IRQn {
4      /* Auxiliary constants */
5      NotAvail_IRQn                = -128,
6
7      /* Core interrupts */
8      Software0_IRQn               = 0,
9      Software1_IRQn               = 1,
10     Software2_IRQn               = 2,
11     Software3_IRQn               = 3,
12     Software4_IRQn               = 4,
13     Software5_IRQn               = 5,
14     Software6_IRQn               = 6,
15     Software7_IRQn               = 7,
16     Software8_IRQn               = 8,
17     Software9_IRQn               = 9,
18     Software10_IRQn              = 10,
19     Software11_IRQn              = 11,
20     Software12_IRQn              = 12,
```

```
21      Software13_IRQn                 = 13,
22      Software14_IRQn                 = 14,
23      Software15_IRQn                 = 15,
24      VirtualMaintenance_IRQn         = 25,
25      HypervisorTimer_IRQn            = 26,
26      VirtualTimer_IRQn               = 27,
27      LegacyFastInt_IRQn              = 28,
28      SecurePhyTimer_IRQn             = 29,
29      NonSecurePhyTimer_IRQn          = 30,
30      LegacyIRQ_IRQn                  = 31,
31
32      /* Device specific interrupts */
33      IOMUXC_IRQn                     = 32,
34      DAP_IRQn                        = 33,
35      SDMA_IRQn                       = 34,
36      TSC_IRQn                        = 35,
37      SNVS_IRQn                       = 36,
……      ......                          ......
151     ENET2_1588_IRQn                 = 153,
152     Reserved154_IRQn                = 154,
153     Reserved155_IRQn                = 155,
154     Reserved156_IRQn                = 156,
155     Reserved157_IRQn                = 157,
156     Reserved158_IRQn                = 158,
157     PMU_IRQ2_IRQn                   = 159
158}    IRQn_Type;
```

3、GIC 逻辑分块

GIC 架构分为了两个逻辑块：Distributor 和 CPU Interface，也就是分发器端和 CPU 接口端。这两个逻辑块的含义如下：

Distributor(分发器端)：从图 17.1.3.2 可以看出，此逻辑块负责处理各个中断事件的分发问题，也就是中断事件应该发送到哪个 CPU Interface 上去。分发器收集所有的中断源，可以控制每个中断的优先级，它总是将优先级最高的中断事件发送到 CPU 接口端。分发器端要做的主要工作如下：

①、全局中断使能控制。

②、控制每一个中断的使能或者关闭。

③、设置每个中断的优先级。

④、设置每个中断的目标处理器列表。

⑤、设置每个外部中断的触发模式：电平触发或边沿触发。

⑥、设置每个中断属于组 0 还是组 1。

CPU Interface(CPU 接口端)：CPU 接口端听名字就知道是和 CPU Core 相连接的，因此在图 17.1.3.2 中每个 CPU Core 都可以在 GIC 中找到一个与之对应的 CPU Interface。CPU 接口端就是分发器和 CPU Core 之间的桥梁，CPU 接口端主要工作如下：

①、使能或者关闭发送到 CPU Core 的中断请求信号。

②、应答中断。
③、通知中断处理完成。
④、设置优先级掩码，通过掩码来设置哪些中断不需要上报给 CPU Core。
⑤、定义抢占策略。
⑥、当多个中断到来的时候，选择优先级最高的中断通知给 CPU Core。
例程“9_int”中的文件 core_ca7.h 定义了 GIC 结构体，此结构体里面的寄存器分为了分发器端和 CPU 接口端，寄存器定义如下所示：

示例代码 17.1.3.2 GIC 控制器结构体

```
/*
 * GIC 寄存器描述结构体，
 * GIC 分为分发器端和 CPU 接口端
 */
1  typedef struct
2  {
3   /* 分发器端寄存器 */
4    uint32_t RESERVED0[1024];
5    __IOM uint32_t D_CTLR;          /* Offset: 0x1000 (R/W) */
6    __IM  uint32_t D_TYPER;         /* Offset: 0x1004 (R/ ) */
7    __IM  uint32_t D_IIDR;          /* Offset: 0x1008 (R/ ) */
8          uint32_t RESERVED1[29];
9    __IOM uint32_t D_IGROUPR[16];  /* Offset: 0x1080 - 0x0BC (R/W) */
10         uint32_t RESERVED2[16];
11   __IOM uint32_t D_ISENABLER[16];/* Offset: 0x1100 - 0x13C (R/W) */
12         uint32_t RESERVED3[16];
13   __IOM uint32_t D_ICENABLER[16];/* Offset: 0x1180 - 0x1BC (R/W) */
14         uint32_t RESERVED4[16];
15   __IOM uint32_t D_ISPENDR[16];  /* Offset: 0x1200 - 0x23C (R/W) */
16         uint32_t RESERVED5[16];
17   __IOM uint32_t D_ICPENDR[16];  /* Offset: 0x1280 - 0x2BC (R/W) */
18         uint32_t RESERVED6[16];
19   __IOM uint32_t D_ISACTIVER[16];/* Offset: 0x1300 - 0x33C (R/W) */
20         uint32_t RESERVED7[16];
21   __IOM uint32_t D_ICACTIVER[16];/* Offset: 0x1380 - 0x3BC (R/W) */
22         uint32_t RESERVED8[16];
23   __IOM uint8_t  D_IPRIORITYR[512];/* Offset: 0x1400 - 0x5FC (R/W) */
24         uint32_t RESERVED9[128];
25   __IOM uint8_t  D_ITARGETSR[512];/* Offset: 0x1800 - 0x9FC (R/W) */
26         uint32_t RESERVED10[128];
27   __IOM uint32_t D_ICFGR[32];     /* Offset: 0x1C00 - 0xC7C (R/W) */
28         uint32_t RESERVED11[32];
29   __IM  uint32_t D_PPISR;         /* Offset: 0x1D00 (R/ ) */
30   __IM  uint32_t D_SPISR[15];     /* Offset: 0x1D04 - 0xD3C (R/ ) */
31         uint32_t RESERVED12[112];
```

```
32   __OM  uint32_t D_SGIR;          /* Offset: 0x1F00 ( /W) */
33         uint32_t RESERVED13[3];
34   __IOM uint8_t  D_CPENDSGIR[16];/* Offset: 0x1F10 - 0xF1C (R/W) */
35   __IOM uint8_t  D_SPENDSGIR[16];/* Offset: 0x1F20 - 0xF2C (R/W) */
36         uint32_t RESERVED14[40];
37   __IM  uint32_t D_PIDR4;         /* Offset: 0x1FD0 (R/ ) */
38   __IM  uint32_t D_PIDR5;         /* Offset: 0x1FD4 (R/ ) */
39   __IM  uint32_t D_PIDR6;         /* Offset: 0x1FD8 (R/ ) */
40   __IM  uint32_t D_PIDR7;         /* Offset: 0x1FDC (R/ ) */
41   __IM  uint32_t D_PIDR0;         /* Offset: 0x1FE0 (R/ ) */
42   __IM  uint32_t D_PIDR1;         /* Offset: 0x1FE4 (R/ ) */
43   __IM  uint32_t D_PIDR2;         /* Offset: 0x1FE8 (R/ ) */
44   __IM  uint32_t D_PIDR3;         /* Offset: 0x1FEC (R/ ) */
45   __IM  uint32_t D_CIDR0;         /* Offset: 0x1FF0 (R/ ) */
46   __IM  uint32_t D_CIDR1;         /* Offset: 0x1FF4 (R/ ) */
47   __IM  uint32_t D_CIDR2;         /* Offset: 0x1FF8 (R/ ) */
48   __IM  uint32_t D_CIDR3;         /* Offset: 0x1FFC (R/ ) */
49 
50   /* CPU 接口端寄存器 */
51   __IOM uint32_t C_CTLR;          /* Offset: 0x2000 (R/W) */
52   __IOM uint32_t C_PMR;           /* Offset: 0x2004 (R/W) */
53   __IOM uint32_t C_BPR;           /* Offset: 0x2008 (R/W) */
54   __IM  uint32_t C_IAR;           /* Offset: 0x200C (R/ ) */
55   __OM  uint32_t C_EOIR;          /* Offset: 0x2010 ( /W) */
56   __IM  uint32_t C_RPR;           /* Offset: 0x2014 (R/ ) */
57   __IM  uint32_t C_HPPIR;         /* Offset: 0x2018 (R/ ) */
58   __IOM uint32_t C_ABPR;          /* Offset: 0x201C (R/W) */
59   __IM  uint32_t C_AIAR;          /* Offset: 0x2020 (R/ ) */
60   __OM  uint32_t C_AEOIR;         /* Offset: 0x2024 ( /W) */
61   __IM  uint32_t C_AHPPIR;        /* Offset: 0x2028 (R/ ) */
62         uint32_t RESERVED15[41];
63   __IOM uint32_t C_APR0;          /* Offset: 0x20D0 (R/W) */
64         uint32_t RESERVED16[3];
65   __IOM uint32_t C_NSAPR0;        /* Offset: 0x20E0 (R/W) */
66         uint32_t RESERVED17[6];
67   __IM  uint32_t C_IIDR;          /* Offset: 0x20FC (R/ ) */
68         uint32_t RESERVED18[960];
69   __OM  uint32_t C_DIR;           /* Offset: 0x3000 ( /W) */
70 } GIC_Type;
```

“示例代码 17.1.3.2”中的结构体 GIC_Type 就是 GIC 控制器，列举除了 GIC 控制器的所有寄存器，可以通过结构体 GIC_Type 来访问 GIC 的所有寄存器。

第 5 行是 GIC 的分发器端相关寄存器，其相对于 GIC 基地址偏移为 0X1000，因此我们获取到 GIC 基地址以后只需要加上 0X1000 即可访问 GIC 分发器端寄存器。

第 51 行是 GIC 的 CPU 接口端相关寄存器，其相对于 GIC 基地址的偏移为 0X2000，同样的，获取到 GIC 基地址以后只需要加上 0X2000 即可访问 GIC 的 CPU 接口段寄存器。
那么问题来了？GIC 控制器的寄存器基地址在哪里呢？这个就需要用到 Cortex-A 的 CP15 协处理器了，下一小节就讲解 CP15 协处理器。

17.1.4 CP15 协处理器

关于 CP15 协处理器和其相关寄存器的详细内容请参考下面两份文档：
《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》第 1469 页“B3.17 Oranization of the CP15 registers in a VMSA implementation”。
《Cortex-A7 Technical ReferenceManua.pdf》第 55 页“Capter 4 System Control”。

CP15 协处理器一般用于存储系统管理，但是在中断中也会使用到，CP15 协处理器一共有 16 个 32 位寄存器。CP15 协处理器的访问通过如下另个指令完成：

MRC: 将 CP15 协处理器中的寄存器数据读到 ARM 寄存器中。

MCR: 将 ARM 寄存器的数据写入到 CP15 协处理器寄存器中。

MRC 就是读 CP15 寄存器，MCR 就是写 CP15 寄存器，MCR 指令格式如下：

```
MCR{cond} p15, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>
```

cond:指令执行的条件码，如果忽略的话就表示无条件执行。

opc1：协处理器要执行的操作码。

Rt：ARM 源寄存器，要写入到 CP15 寄存器的数据就保存在此寄存器中。

CRn：CP15 协处理器的目标寄存器。

CRm：协处理器中附加的目标寄存器或者源操作数寄存器，如果不需要附加信息就将 CRm 设置为 C0，否则结果不可预测。

opc2：可选的协处理器特定操作码，当不需要的时候要设置为 0。

MRC 的指令格式和 MCR 一样，只不过在 MRC 指令中 Rt 就是目标寄存器，也就是从 CP15 指定寄存器读出来的数据会保存在 Rt 中。而 CRn 就是源寄存器，也就是要读取的写处理器寄存器。

假如我们要将 CP15 中 C0 寄存器的值读取到 R0 寄存器中，那么就可以使用如下命令：

```
MRC p15, 0, r0, c0, c0, 0
```

CP15 协处理器有 16 个 32 位寄存器，c0~c15，本章来看一下 c0、c1、c12 和 c15 这四个寄存器，因为我们本章实验要用到这四个寄存器，其他的寄存器大家参考上面的两个文档即可。

1、c0 寄存器

CP15 协处理器有 16 个 32 位寄存器，c0~c15，在使用 MRC 或者 MCR 指令访问这 16 个寄存器的时候，指令中的 CRn、opc1、CRm 和 opc2 通过不同的搭配，其得到的寄存器含义是不同的。比如 c0 在不同的搭配情况下含义如图 17.1.4.1 所示：

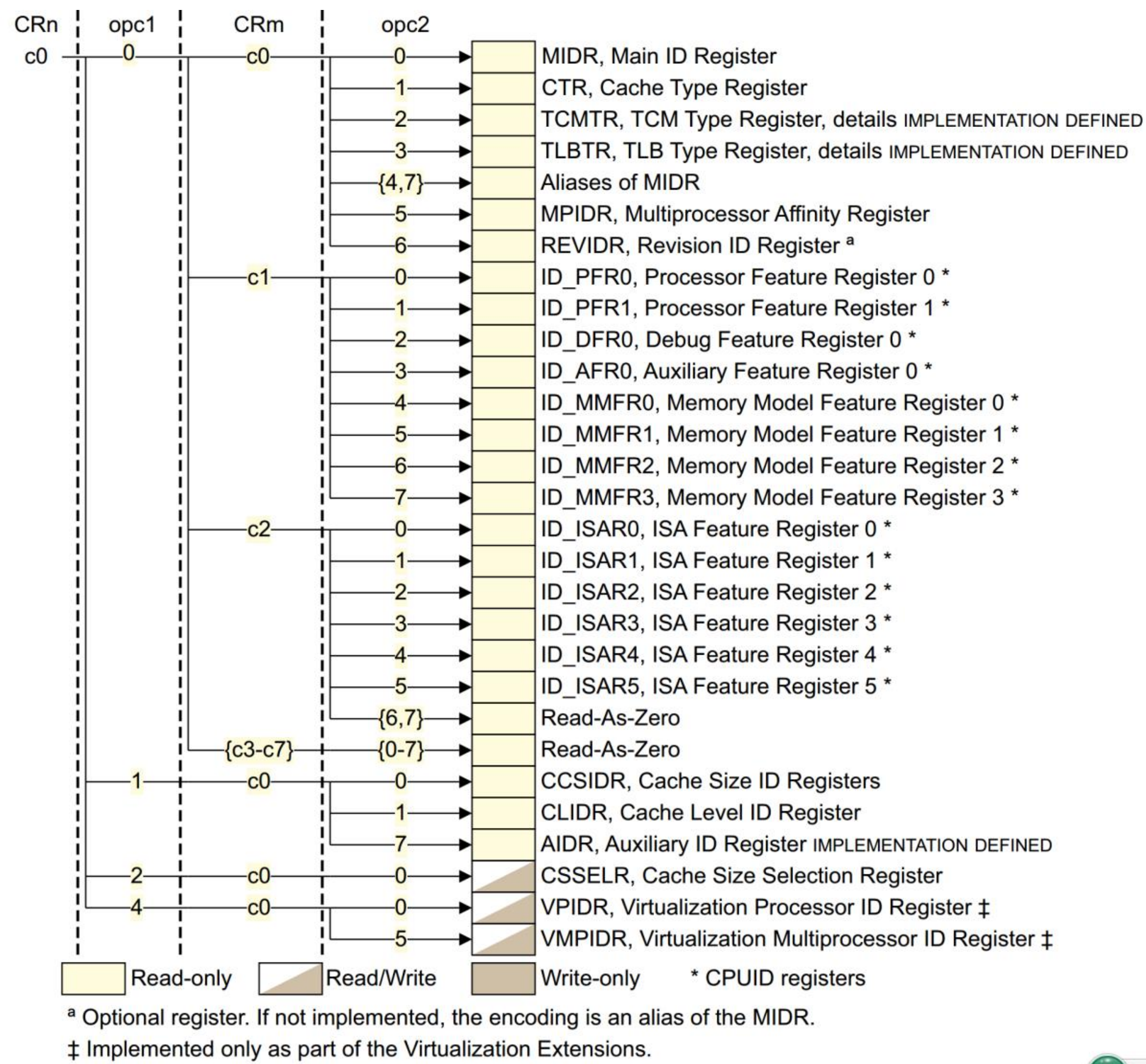


图 17.1.4.1 c0 寄存器不同搭配含义

在图 17.1.4.1 中当 MRC/MCR 指令中的 CRn=c0，opc1=0，CRm=c0，opc2=0 的时候就表示此时的 c0 就是 MIDR 寄存器，也就是主 ID 寄存器，这个也是 c0 的基本作用。对于 Cortex-A7 内核来说，c0 作为 MDIR 寄存器的时候其含义如图 17.1.4.2 所示：

31　　24	23　　20	19　　16	15　　4	3　　0
Implementer	Variant	Architecture	Primary part number	Revision

图 17.1.4.2 c0 作为 MIDR 寄存器结构图

在图 17.1.4.2 中各位所代表的含义如下：

bit31:24：厂商编号，0X41，ARM。

bit23:20：内核架构的主版本号，ARM 内核版本一般使用 rnpn 来表示，比如 r0p1，其中 r0 后面的 0 就是内核架构主版本号。

bit19:16：架构代码，0XF，ARMv7 架构。

bit15:4：内核版本号，0XC07，Cortex-A7 MPCore 内核。

bit3:0：内核架构的次版本号，rnpn 中的 pn，比如 r0p1 中 p1 后面的 1 就是次版本号。

2、c1 寄存器

c1 寄存器同样通过不同的配置，其代表的含义也不同，如图 17.1.4.3 所示：

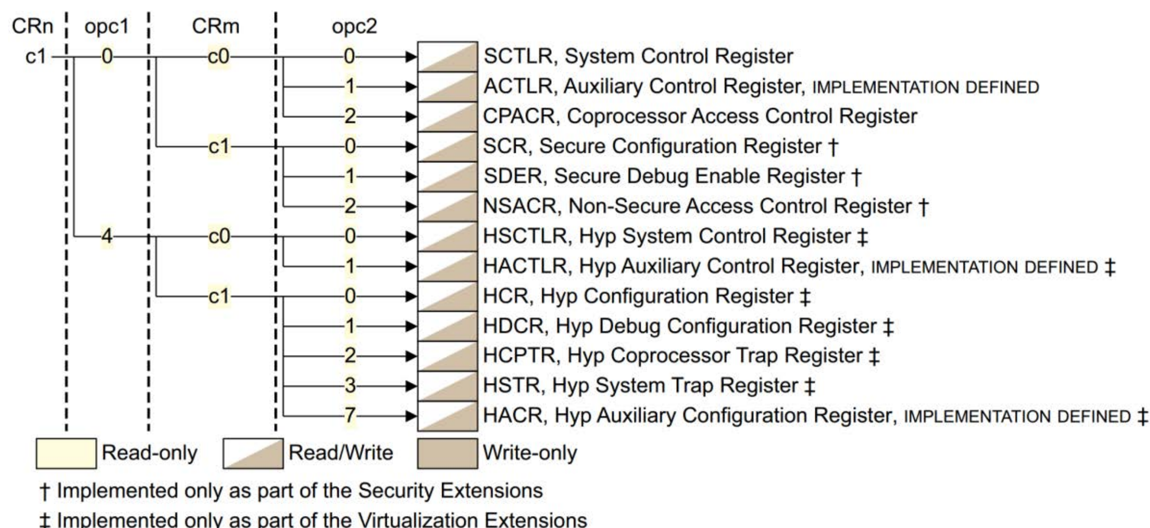


图 17.1.4.3 c1 寄存器不同搭配含义

在图 17.1.4.3 中当 MRC/MCR 指令中的 CRn=c1，opc1=0，CRm=c0，opc2=0 的时候就表示此时的 c1 就是 SCTLR 寄存器，也就是系统控制寄存器，这个是 c1 的基本作用。SCTLR 寄存器主要是完成控制功能的，比如使能或者禁止 MMU、I/D Cache 等，c1 作为 SCTLR 寄存器的时候其含义如图 17.1.4.4 所示：

30	31	29	28	27 26	25	24 21	20	19	18 14	13	12	11	10	9 3	2	1	0
Res	TE	AFE	TRE	Res	EE	Res	UWXN	WXN	Res	V	I	Z	SW	Res	C	A	M

图 17.1.4.4 c1 作为 SCTLR 寄存器结构图

SCTLR 的位比较多，我们就只看本章会用到的几个位：

bit13：V，中断向量表基地址选择位，为 0 的话中断向量表基地址为 0X00000000，软件可以使用 VBAR 来重映射此基地址，也就是中断向量表重定位。为 1 的话中断向量表基地址为 0XFFFF0000，此基地址不能被重映射。

bit12：I，I Cache 使能位，为 0 的话关闭 I Cache，为 1 的话使能 I Cache。

bit11：Z，分支预测使能位，如果开启 MMU 的话，此为也会使能。

bit10：SW，SWP 和 SWPB 使能位，当为 0 的话关闭 SWP 和 SWPB 指令，当为 1 的时候就使能 SWP 和 SWPB 指令。

bit9:3：未使用，保留。

bit2：C，D Cache 和缓存一致性使能位，为 0 的时候禁止 D Cache 和缓存一致性，为 1 时使能。

bit1：A，内存对齐检查使能位，为 0 的时候关闭内存对齐检查，为 1 的时候使能内存对齐检查。

bit0：M，MMU 使能位，为 0 的时候禁止 MMU，为 1 的时候使能 MMU。

如果要读写 SCTLR 的话，就可以使用如下命令：

```
MRC p15, 0, <Rt>, c1, c0, 0    ;读取 SCTLR 寄存器，数据保存到 Rt 中。
MCR p15, 0, <Rt>, c1, c0, 0    ;将 Rt 中的数据写到 SCTLR(c1)寄存器中。
```

2、c12 寄存器

c12 寄存器通过不同的配置，其代表的含义也不同，如图 17.1.4.4 所示：

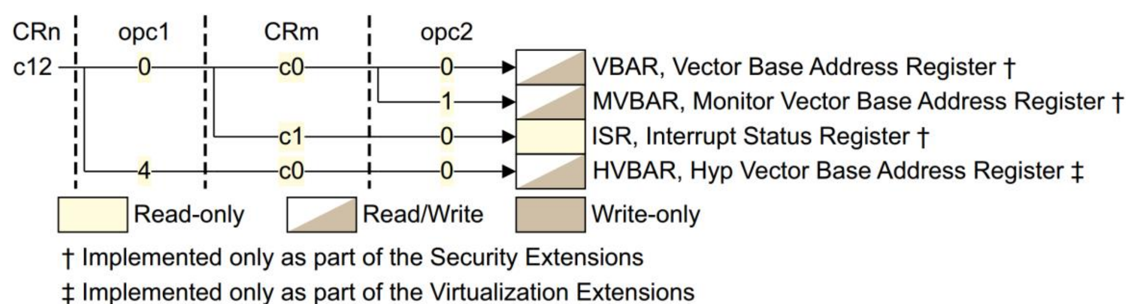


图 17.1.4.4　c12 寄存器不同搭配含义

在图 17.1.4.4 中当 MRC/MCR 指令中的 CRn=c12，opc1=0，CRm=c0，opc2=0 的时候就表示此时 c12 为 VBAR 寄存器，也就是向量表基地址寄存器。设置中断向量表偏移的时候就需要将新的中断向量表基地址写入 VBAR 中，比如在前面的例程中，代码链接的起始地址为 0X87800000，而中断向量表肯定要放到最前面，也就是 0X87800000 这个地址处。所以就需要设置 VBAR 为 0X87800000，设置命令如下：

```
ldr r0, =0X87800000              ; r0=0X87800000
MCR p15, 0, r0, c12, c0, 0       ;将 r0 里面的数据写入到 c12 中，即 c12=0X87800000
```

3、c15 寄存器

c15 寄存器也可以通过不同的配置得到不同的含义，参考文档《Cortex-A7 Technical ReferenceManua.pdf》第 68 页“4.2.16 c15 registers”，其配置如图 17.1.4.5 所示：

CRn	Op1	CRm	Op2	Name	Reset	Description
c15	3[a]	c0	0	CDBGDR0	UNK	Data Register 0, see *Direct access to internal memory* on page 6-9
			1	CDBGDR1	UNK	Data Register 1, see *Direct access to internal memory* on page 6-9
			2	CDBGDR2	UNK	Data Register 2, see *Direct access to internal memory* on page 6-9
		c2	0	CDBGDCT	UNK	Data Cache Tag Read Operation Register, see *Direct access to internal memory* on page 6-9
			1	CDBGICT	UNK	Instruction Cache Tag Read Operation Register, see *Direct access to internal memory* on page 6-9
		c4	0	CDBGDCD	UNK	Data Cache Data Read Operation Register, see *Direct access to internal memory* on page 6-9
			1	CDBGICD	UNK	Instruction Cache Data Read Operation Register, see *Direct access to internal memory* on page 6-9
			2	CDBGTD	UNK	TLB Data Read Operation Register, see *Direct access to internal memory* on page 6-9
	4	c0	0	CBAR	-[b]	*Configuration Base Address Register* on page 4-83

图 17.1.4.5 c15 寄存器不同搭配含义

在图 17.1.4.5 中，我们需要 c15 作为 CBAR 寄存器，因为 GIC 的基地址就保存在 CBAR 中，我们可以通过如下命令获取到 GIC 基地址：

```
MRC p15, 4, r1, c15, c0, 0 ;  获取 GIC 基础地址，基地址保存在 r1 中。
```

获取到 GIC 基地址以后就可以设置 GIC 相关寄存器了，比如我们可以读取当前中断 ID，当前中断 ID 保存在 GICC_IAR 中，寄存器 GICC_IAR 属于 CPU 接口端寄存器，寄存器地址相对于 CPU 接口端起始地址的偏移为 0XC，因此获取当前中断 ID 的代码如下：

```
MRC p15, 4, r1, c15, c0, 0    ;获取 GIC 基地址
ADD r1, r1, #0X2000           ;GIC 基地址加 0X2000 得到 CPU 接口端寄存器起始地址
```

```
LDR r0, [r1, #0XC]          ;读取 CPU 接口端起始地址+0XC 处的寄存器值，也就是寄存器
                            ;GIC_IAR 的值
```

关于 CP15 协处理器就讲解到这里，简单总结一下，通过 c0 寄存器可以获取到处理器内核信息；通过 c1 寄存器可以使能或禁止 MMU、I/D Cache 等；通过 c12 寄存器可以设置中断向量偏移；通过 c15 寄存器可以获取 GIC 基地址。关于 CP15 的其他寄存器，大家自行查阅本节前面列举的 2 份 ARM 官方资料。

17.1.5 中断使能

中断使能包括两部分，一个是 IRQ 或者 FIQ 总中断使能，另一个就是 ID0~ID1019 这 1020 个中断源的使能。

1、IRQ 和 FIQ 总中断使能

IRQ 和 FIQ 分别是外部中断和快速中断的总开关，就类似家里买的进户总电闸，然后 ID0~ID1019 这 1020 个中断源就类似家里面的各个电器开关。要想开电视，那肯定要保证进户总电闸是打开的，因此要想使用 I.MX6U 上的外设中断就必须先打开 IRQ 中断(本教程不使用 FIQ)。在“6.3.2 程序状态寄存器”小节已经讲过了，寄存器 CPSR 的 I=1 禁止 IRQ，当 I=0 使能 IRQ；F=1 禁止 FIQ，F=0 使能 FIQ。我们还有更简单的指令来完成 IRQ 或者 FIQ 的使能和禁止，图表 17.1.5.1 所示：

指令	描述
cpsid i	禁止 IRQ 中断。
cpsie i	使能 IRQ 中断。
cpsid f	禁止 FIQ 中断。
cpsie f	使能 FIQ 中断。

表 17.1.5.1 开关中断指令

2、ID0~ID1019 中断使能和禁止

GIC 寄存器 GICD_ISENABLERn 和 GICD_ ICENABLERn 用来完成外部中断的使能和禁止，对于 Cortex-A7 内核来说中断 ID 只使用了 512 个。一个 bit 控制一个中断 ID 的使能，那么就需要 512/32=16 个 GICD_ISENABLER 寄存器来完成中断的使能。同理，也需要 16 个 GICD_ICENABLER 寄存器来完成中断的禁止。其中 GICD_ISENABLER0 的 bit[15:0]对应 ID15~0 的 SGI 中断，GICD_ISENABLER0 的 bit[31:16]对应 ID31~16 的 PPI 中断。剩下的 GICD_ISENABLER1~GICD_ISENABLER15 就是控制 SPI 中断的。

17.1.6 中断优先级设置

1、优先级数配置

学过 STM32 都知道 Cortex-M 的中断优先级分为抢占优先级和子优先级，两者是可以配置的。同样的 Cortex-A7 的中断优先级也可以分为抢占优先级和子优先级，两者同样是可以配置的。Cortex-A7 最多可以支持 256 个优先级，数字越小，优先级越高！半导体厂商自行决定选择多少个优先级。I.MX6U 选择了 32 个优先级。在使用中断的时候需要初始化 GICC_PMR 寄存器，此寄存器用来决定使用几级优先级，寄存器结构如图 17.1.6.1 所示：



图 17.1.6.1 GICC_PMR 寄存器

GICC_PMR 寄存器只有低 8 位有效，这 8 位最多可以设置 256 个优先级，其他优先级数设置如表 17.1.6.1 所示：

bit7:0	优先级数
11111111	256 个优先级。
11111110	128 个优先级。
11111100	64 个优先级。
11111000	32 个优先级
11110000	16 个优先级。

表 17.1.6.1 优先级数设置

I.MX6U 支持 32 个优先级，所以 GICC_PMR 要设置为 0b11111000。

2、抢占优先级和子优先级位数设置

抢占优先级和子优先级各占多少位是由寄存器 GICC_BPR 来决定的，GICC_BPR 寄存器结构如图 17.1.6.2 所示：



图 17.1.6.2 GICC_BPR 寄存器结构图

寄存器 GICC_BPR 只有低 3 位有效，其值不同，抢占优先级和子优先级占用的位数也不同，配置如表 17.1.6.2 所示：

Binary Point	抢占优先级域	子优先级域	描述
0	[7:1]	[0]	7 级抢占优先级，1 级子优先级。
1	[7:2]	[1:0]	6 级抢占优先级，2 级子优先级。
2	[7:3]	[2:0]	5 级抢占优先级，3 级子优先级。
3	[7:4]	[3:0]	4 级抢占优先级，4 级子优先级。
4	[7:5]	[4:0]	3 级抢占优先级，5 级子优先级。
5	[7:6]	[5:0]	2 级抢占优先级，6 级子优先级。
6	[7:7]	[6:0]	1 级抢占优先级，7 级子优先级。
7	无	[7:0]	0 级抢占优先级，8 级子优先级。

表 17.1.6.2 GICC_BPR 配置表

为了简单起见，一般将所有的中断优先级位都配置为抢占优先级，比如 I.MX6U 的优先级位数为 5(32 个优先级)，所以可以设置 Binary point 为 2，表示 5 个优先级位全部为抢占优先级。

3、优先级设置

前面已经设置好了 I.MX6U 一共有 32 个抢占优先级，数字越小优先级越高。具体要使用某个中断的时候就可以设置其优先级为 0~31。某个中断 ID 的中断优先级设置由寄存器 D_IPRIORITYR 来完成，前面说了 Cortex-A7 使用了 512 个中断 ID，每个中断 ID 配有一个优先级寄存器，所以一共有 512 个 D_IPRIORITYR 寄存器。如果优先级个数为 32 的话，使用寄存器 D_IPRIORITYR 的 bit7:4 来设置优先级，也就是说实际的优先级要左移 3 位。比如要设置 ID40 中断的优先级为 5，示例代码如下：

```
GICD_IPRIORITYR[40] = 5 << 3;
```

有关优先级设置的内容就讲解到这里，优先级设置主要有三部分：

①、设置寄存器 GICC_PMR，配置优先级个数，比如 I.MX6U 支持 32 级优先级。

②、设置抢占优先级和子优先级位数，一般为了简单起见，会将所有的位数都设置为抢占优先级。

③、设置指定中断 ID 的优先级，也就是设置外设优先级。

17.2 硬件原理分析

本试验用到的硬件资源和第十五章的硬件资源一模一样。

17.3 试验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 9_int。

本章试验的功能和第十五章一样，只是按键采用中断的方式处理。当按下按键 KEY0 以后就打开蜂鸣器，再次按下按键 KEY0 就关闭蜂鸣器。在第十六章的试验上完成本章试验。

17.3.1 移植 SDK 包中断相关文件

将 SDK 包中的文件 core_ca7.h 拷贝到本章试验工程中的“imx6ul”文件夹中，参考试验“9_int”中 core_ca7.h 进行修改。主要留下和 GIC 相关的内容，我们重点是需要 core_ca7.h 中的 10 个 API 函数，这 10 个函数如表 17.3.1.1 所示：

函数	描述
GIC_Init	初始化 GIC。
GIC_EnableIRQ	使能指定的外设中断。
GIC_DisableIRQ	关闭指定的外设中断。
GIC_AcknowledgeIRQ	返回中断号。
GIC_DeactivateIRQ	无效化指定中断。
GIC_GetRunningPriority	获取当前正在运行的中断优先级。
GIC_SetPriorityGrouping	设置抢占优先级位数。
GIC_GetPriorityGrouping	获取抢占优先级位数。
GIC_SetPriority	设置指定中断的优先级。
GIC_GetPriority	获取指定中断的优先级。

表 17.3.1.1 GIC 相关 API 操作函数

移植好 core_ca7.h 以后，修改文件 imx6ul.h，在里面加上如下一行代码：

```
#include "core_ca7.h"
```

17.3.2 重新编写 start.S 文件

重新在 start.S 中输入如下内容：

示例代码 17.3.2.1 start.S 文件代码

```
/*****************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : start.s
作者      : 左忠凯
版本      : V2.0
描述      : I.MX6U-ALPHA/I.MX6ULL 开发板启动文件，完成 C 环境初始化，
            C 环境初始化完成以后跳转到 C 代码。
其他      : 无
```

```
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/3 左忠凯修改
            V2.0 2019/1/4 左忠凯修改
            添加中断相关定义
**************************************************************/

1  .global _start              /* 全局标号 */
2
3  /*
4   * 描述： _start 函数，首先是中断向量表的创建
5   */
6  _start:
7       ldr pc, =Reset_Handler      /* 复位中断              */
8       ldr pc, =Undefined_Handler  /* 未定义指令中断        */
9       ldr pc, =SVC_Handler        /* SVC(Supervisor)中断*/
10      ldr pc, =PrefAbort_Handler  /* 预取终止中断          */
11      ldr pc, =DataAbort_Handler  /* 数据终止中断          */
12      ldr pc, =NotUsed_Handler    /* 未使用中断            */
13      ldr pc, =IRQ_Handler        /* IRQ 中断              */
14      ldr pc, =FIQ_Handler        /* FIQ(快速中断)         */
15
16 /* 复位中断 */
17 Reset_Handler:
18
19      cpsid i                     /* 关闭全局中断 */
20
21    /* 关闭 I,DCache 和 MMU
22     * 采取读-改-写的方式。
23     */
24     mrc     p15, 0, r0, c1, c0, 0 /* 读取 CP15 的 C1 寄存器到 R0 中  */
25     bic     r0,  r0, #(0x1 << 12) /* 清除 C1 的 I 位，关闭 I Cache  */
26     bic     r0,  r0, #(0x1 <<  2) /* 清除 C1 的 C 位，关闭 D Cache  */
27     bic     r0,  r0, #0x2         /* 清除 C1 的 A 位，关闭对齐检查  */
28     bic     r0,  r0, #(0x1 << 11) /* 清除 C1 的 Z 位，关闭分支预测  */
29     bic     r0,  r0, #0x1         /* 清除 C1 的 M 位，关闭 MMU      */
30     mcr     p15, 0, r0, c1, c0, 0 /* 将 r0 的值写入到 CP15 的 C1 中  */
31
32
33 #if 0
34      /* 汇编版本设置中断向量表偏移 */
35      ldr r0, =0X87800000
36
37      dsb
```

```
38      isb
39      mcr p15, 0, r0, c12, c0, 0
40      dsb
41      isb
42  #endif
43
44      /* 设置各个模式下的栈指针，
45       * 注意：IMX6UL 的堆栈是向下增长的！
46       * 堆栈指针地址一定要是 4 字节地址对齐的！！！
47       * DDR 范围:0X80000000~0X9FFFFFFF 或者 0X8FFFFFFF
48       */
49      /* 进入 IRQ 模式 */
50      mrs r0, cpsr
51      bic r0, r0, #0x1f     /* 将 r0 的低 5 位清零，也就是 cpsr 的 M0~M4   */
52      orr r0, r0, #0x12     /* r0 或上 0x12,表示使用 IRQ 模式              */
53      msr cpsr, r0          /* 将 r0 的数据写入到 cpsr 中                  */
54      ldr sp, =0x80600000   /* IRQ 模式栈首地址为 0X80600000,大小为 2MB */
55
56      /* 进入 SYS 模式 */
57      mrs r0, cpsr
58      bic r0, r0, #0x1f     /* 将 r0 的低 5 位清零，也就是 cpsr 的 M0~M4   */
59      orr r0, r0, #0x1f     /* r0 或上 0x13,表示使用 SYS 模式              */
60      msr cpsr, r0          /* 将 r0 的数据写入到 cpsr 中                  */
61      ldr sp, =0x80400000   /* SYS 模式栈首地址为 0X80400000,大小为 2MB */
62
63      /* 进入 SVC 模式 */
64      mrs r0, cpsr
65      bic r0, r0, #0x1f     /* 将 r0 的低 5 位清零，也就是 cpsr 的 M0~M4   */
66      orr r0, r0, #0x13     /* r0 或上 0x13,表示使用 SVC 模式              */
67      msr cpsr, r0          /* 将 r0 的数据写入到 cpsr 中                  */
68      ldr sp, =0X80200000   /* SVC 模式栈首地址为 0X80200000,大小为 2MB */
69
70      cpsie i               /* 打开全局中断 */
71
72   #if 0
73      /* 使能 IRQ 中断 */
74      mrs r0, cpsr          /* 读取 cpsr 寄存器值到 r0 中                  */
75      bic r0, r0, #0x80     /* 将 r0 寄存器中 bit7 清零，也就是 CPSR 中
76                             * 的 I 位清零，表示允许 IRQ 中断
77                             */
78      msr cpsr, r0          /* 将 r0 重新写入到 cpsr 中                    */
79   #endif
80
```

```
81      b main                  /* 跳转到 main 函数                     */
82
83  /* 未定义中断 */
84  Undefined_Handler:
85      ldr r0, =Undefined_Handler
86      bx r0
87
88  /* SVC 中断 */
89  SVC_Handler:
90      ldr r0, =SVC_Handler
91      bx r0
92
93  /* 预取终止中断 */
94  PrefAbort_Handler:
95      ldr r0, =PrefAbort_Handler
96      bx r0
97
98  /* 数据终止中断 */
99  DataAbort_Handler:
100     ldr r0, =DataAbort_Handler
101     bx r0
102
103 /* 未使用的中断 */
104 NotUsed_Handler:
105
106     ldr r0, =NotUsed_Handler
107     bx r0
108
109 /* IRQ 中断！重点！！！！！ */
110 IRQ_Handler:
111     push {lr}               /* 保存 lr 地址              */
112     push {r0-r3, r12}       /* 保存 r0-r3，r12 寄存器    */
113
114     mrs r0, spsr            /* 读取 spsr 寄存器          */
115     push {r0}               /* 保存 spsr 寄存器          */
116
117     mrc p15, 4, r1, c15, c0, 0 /* 将 CP15 的 C0 内的值到 R1 寄存器中
118                         * 参考文档 ARM Cortex-A(armV7)编程手册 V4.0.pdf P49
119                         * Cortex-A7 Technical ReferenceManua.pdf P68 P138
120                         */
121     add r1, r1, #0X2000     /* GIC 基地址加 0X2000，得到 CPU 接口端基地址 */
122     ldr r0, [r1, #0XC]      /* CPU 接口端基地址加 0X0C 就是 GICC_IAR 寄存器，
123                              * GICC_IAR 保存着当前发生中断的中断号，我们要根据
```

```
124                          * 这个中断号来绝对调用哪个中断服务函数
125                            */
126     push {r0, r1}         /* 保存 r0,r1                    */
127
128     cps #0x13             /* 进入 SVC 模式，允许其他中断再次进去    */
129
130     push {lr}             /* 保存 SVC 模式的 lr 寄存器           */
131     ldr r2, =system_irqhandler  /* 加载 C 语言中断处理函数到 r2 寄存器中*/
132     blx r2                /* 运行 C 语言中断处理函数，带有一个参数  */
133
134     pop {lr}              /* 执行完 C 语言中断服务函数，lr 出栈     */
135     cps #0x12             /* 进入 IRQ 模式                    */
136     pop {r0, r1}
137     str r0, [r1, #0X10]   /* 中断执行完成，写 EOIR              */
138
139     pop {r0}
140     msr spsr_cxsf, r0     /* 恢复 spsr                       */
141
142     pop {r0-r3, r12}      /* r0-r3,r12 出栈                  */
143     pop {lr}              /* lr 出栈                         */
144     subs pc, lr, #4       /* 将 lr-4 赋给 pc                  */
145
146 /* FIQ 中断 */
147 FIQ_Handler:
148
149     ldr r0, =FIQ_Handler
150     bx r0
```

第 6 到 14 行是中断向量表，17.1.2 小节已经讲解过了。

第 17 到 81 行是复位中断服务函数 Reset_Handler， 第 19 行先调用指令“cpsid i”关闭 IRQ，第 24 到 30 行是关闭 I/D Cache、MMU、对齐检测和分支预测。第 33 行到 42 行是汇编版本的中断向量表重映射。第 50 到 68 行是设置不同模式下的 sp 指针，分别设置 IRQ 模式、SYS 模式和 SVC 模式的栈指针，每种模式的栈大小都是 2MB。第 70 行调用指令“cpsie i”重新打开 IRQ 中断，第 72 到 79 行是操作 CPSR 寄存器来打开 IRQ 中断。当初始化工作都完成以后就可以进入到 main 函数了，第 81 行就是跳转到 main 函数。

第 110 到 144 行是中断服务函数 IRQ_Handler，这个是本章的重点，因为所有的外部中断最终都会触发 IRQ 中断，所以 IRQ 中断服务函数主要的工作就是区分去当前发生的什么中断(中断 ID)？然后针对不同的外部中断做出不同的处理。第 111 到 115 行是保存现场，第 117 到 122 行是获取当前中断号，中断号被保存到了 r0 寄存器中。第 131 和 132 行才是中断处理的重点，这两行相当于调用了函数 system_irqhandler，函数 system_irqhandler 是一个 C 语言函数，此函数有一个参数，这个参数中断号，所以我们需要传递一个参数。汇编中调用 C 函数如何实现参数传递呢？根据 ATPCS(ARM-Thumb Procedure Call Standard)定义的函数参数传递规则，在汇编调用 C 函数的时候建议形参不要超过 4 个，形参可以由 r0~r3 这四个寄存器来传递，如果形参大于 4 个，那么大于 4 个的部分要使用堆栈进行传递。所以给 r0 寄存器写入中断号就可以

了函数 system_irqhandler 的参数传递，在 136 行已经向 r0 寄存器写入了中断号了。中断的真正处理过程其实是在函数 system_irqhandler 中完成，稍后需要编写函数 stem_irqhandler。

第 151 行向 GICC_EOIR 寄存器写入刚刚处理完成的中断号，当一个中断处理完成以后必须向 GICC_EOIR 寄存器写入其中断号表示中断处理完成。

第 153 到 157 行就是恢复现场。

第 158 行中断处理完成以后就要重新返回到曾经被中断打断的地方运行，这里为什么要将 lr-4 然后赋给 pc 呢？而不是直接将 lr 赋值给 pc？ARM 的指令是三级流水线：取指、译指、执行，pc 指向的是正在取值的地址，这就是很多书上说的 pc=当前执行指令地址+8。比如下面代码示例：

```
0X2000  MOV R1, R0  ;执行
0X2004  MOV R2, R3  ;译指
0X2008  MOV R4, R5  ;取值 PC
```

上面示例代码中，左侧一列是地址，中间是指令，最右边是流水线。当前正在执行 0X2000 地址处的指令“MOV R1, R0”，但是 PC 里面已经保存了 0X2008 地址处的指令“MOV R4, R5”。假设此时发生了中断，中断发生的时候保存在 lr 中的是 pc 的值，也就是地址 0X2008。当中断处理完成以后肯定需要回到被中断点接着执行，如果直接跳转到 lr 里面保存的地址处(0X2008)开始运行，那么就有一个指令没有执行，那就是地址 0X2004 处的指令“MOV R2, R3”，显然这是一个很严重的错误！所以就需要将 lr-4 赋值给 pc，也就是 pc=0X2004，从指令“MOV R2，R3”开始执行。

17.3.3 通用中断驱动文件编写

在 start.S 文件中我们在中断服务函数 IRQ_Handler 中调用了 C 函数 system_irqhandler 来处理具体的中断。此函数有一个参数，参数是中断号，但是函数 system_irqhandler 的具体内容还没有实现，所以需要实现函数 system_irqhandler 的具体内容。不同的中断源对应不同的中断处理函数，I.MX6U 有 160 个中断源，所以需要 160 个中断处理函数，我们可以将这些中断处理函数放到一个数组里面，中断处理函数在数组中的标号就是其对应的中断号。当中断发生以后函数 system_irqhandler 根据中断号从中断处理函数数组中找到对应的中断处理函数并执行即可。

在 bsp 目录下新建名为“int”的文件夹，在 bsp/int 文件夹里面创建 bsp_int.c 和 bsp_int.h 这两个文件。在 bsp_int.h 文件里面输入如下内容：

示例代码 17.3.3.1 bsp_int.h 文件代码

```
1  #ifndef _BSP_INT_H
2  #define _BSP_INT_H
3  #include "imx6ul.h"
4  /***************************************************************
5  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6  文件名    : bsp_int.h
7  作者      : 左忠凯
8  版本      : V1.0
9  描述      : 中断驱动头文件。
10 其他      : 无
11 论坛      : www.openedv.com
12 日志      : 初版 V1.0 2019/1/4 左忠凯创建
13 ***************************************************************/
```

```
14
15 /* 中断处理函数形式 */
16 typedef void (*system_irq_handler_t) (unsigned int giccIar,
                                        void *param);
17
18 /* 中断处理函数结构体*/
19 typedef struct _sys_irq_handle
20 {
21     system_irq_handler_t irqHandler;    /* 中断处理函数 */
22     void *userParam;                    /* 中断处理函数参数 */
23 } sys_irq_handle_t;
24
25 /* 函数声明 */
26 void int_init(void);
27 void system_irqtable_init(void);
28 void system_register_irqhandler(IRQn_Type irq,
                                   system_irq_handler_t handler,
                                   void *userParam);
29 void system_irqhandler(unsigned int giccIar);
30 void default_irqhandler(unsigned int giccIar, void *userParam);
31
32 #endif
```

第 16~23 行是中断处理结构体，结构体 sys_irq_handle_t 包含一个中断处理函数和中断处理函数的用户参数。一个中断源就需要一个 sys_irq_handle_t 变量，I.MX6U 有 160 个中断源，因此需要 160 个 sys_irq_handle_t 组成中断处理数组。

在 bsp_int.c 中输入如下所示代码：

示例代码 17.3.3.2 bsp_int.c 文件代码

```
1  #include "bsp_int.h"
2  /***************************************************************
3  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
4  文件名     : bsp_int.c
5  作者       : 左忠凯
6  版本       : V1.0
7  描述       : 中断驱动文件。
8  其他       : 无
9  论坛       : www.openedv.com
10 日志       : 初版 V1.0 2019/1/4 左忠凯创建
11 ***************************************************************/
12
13 /* 中断嵌套计数器 */
14 static unsigned int irqNesting;
15
16 /* 中断服务函数表 */
```

```
17 static sys_irq_handle_t irqTable[NUMBER_OF_INT_VECTORS];
18 
19 /*
20  * @description : 中断初始化函数
21  * @param       : 无
22  * @return      : 无
23  */
24 void int_init(void)
25 {
26     GIC_Init();                          /* 初始化 GIC        */
27     system_irqtable_init();              /* 初始化中断表      */
28     __set_VBAR((uint32_t)0x87800000);    /* 中断向量表偏移    */
29 }
30 
31 /*
32  * @description  : 初始化中断服务函数表
33  * @param        : 无
34  * @return       : 无
35  */
36 void system_irqtable_init(void)
37 {
38     unsigned int i = 0;
39     irqNesting = 0;
40 
41     /* 先将所有的中断服务函数设置为默认值 */
42     for(i = 0; i < NUMBER_OF_INT_VECTORS; i++)
43     {
44         system_register_irqhandler( (IRQn_Type)i, 
                                       default_irqhandler, 
                                        NULL);
45     }
46 }
47 
48 /*
49  * @description          : 给指定的中断号注册中断服务函数
50  * @param - irq          : 要注册的中断号
51  * @param - handler      : 要注册的中断处理函数
52  * @param - usrParam     : 中断服务处理函数参数
53  * @return               : 无
54  */
55 void system_register_irqhandler(IRQn_Type irq, 
                                 system_irq_handler_t handler, 
                                 void *userParam)
```

```
56 {
57     irqTable[irq].irqHandler = handler;
58     irqTable[irq].userParam = userParam;
59 }
60
61 /*
62  * @description         : C 语言中断服务函数，irq 汇编中断服务函数会
63                            调用此函数，此函数通过在中断服务列表中查
64                            找指定中断号所对应的中断处理函数并执行。
65  * @param - giccIar     : 中断号
66  * @return              : 无
67  */
68 void system_irqhandler(unsigned int giccIar)
69 {
70
71     uint32_t intNum = giccIar & 0x3FFUL;
72
73     /* 检查中断号是否符合要求 */
74     if ((intNum == 1020) || (intNum >= NUMBER_OF_INT_VECTORS))
75     {
76         return;
77     }
78
79     irqNesting++; /* 中断嵌套计数器加一 */
80
81     /* 根据传递进来的中断号，在 irqTable 中调用确定的中断服务函数*/
82     irqTable[intNum].irqHandler(intNum, irqTable[intNum].userParam);
83
84     irqNesting--; /* 中断执行完成，中断嵌套寄存器减一 */
85
86 }
87
88 /*
89  * @description           : 默认中断服务函数
90  * @param - giccIar       : 中断号
91  * @param - usrParam      : 中断服务处理函数参数
92  * @return                : 无
93  */
94 void default_irqhandler(unsigned int giccIar, void *userParam)
95 {
96     while(1)
97     {
98     }
```

```
99 }
```

第 14 行定义了一个变量 irqNesting，此变量作为中断嵌套计数器。

第 17 行定了中断服务函数数组 irqTable，这是一个 sys_irq_handle_t 类型的结构体数组，数组大小为 I.MX6U 的中断源个数，即 160 个。

第 24~28 行是中断初始化函数 int_init，在此函数中首先初始化了 GIC，然后初始化了中断服务函数表，最终设置了中断向量表偏移。

第 36~46 行是中断服务函数表初始化函数 system_irqtable_init，初始化 irqTable，给其赋初值。

第 55~59 行是注册中断处理函数 system_register_irqhandler，此函数用来给指定的中断号注册中断处理函数。如果要使用某个外设中断，那就必须调用此函数来给这个中断注册一个中断处理函数。

第 68~86 行就是前面在 start.S 中调用的 system_irqhandler 函数，此函数根据中断号在中断处理函数表 irqTable 中取出对应的中断处理函数并执行。

第 94~99 行是默认中断处理函数 default_irqhandler，这是一个空函数，主要用来给初始化中断函数处理表。

17.3.4 修改 GPIO 驱动文件

在前几章节试验中我们只是使用到了 GPIO 最基本的输入输出功能，本章我们需要使用 GPIO 的中断功能。所以需要修改文件 GPIO 的驱动文件 bsp_gpio.c 和 bsp_gpio.h，加上中断相关函数。关于 GPIO 中断内容已经在 8.1.5 小节进行了详细的讲解，这里就不赘述了。打开 bsp_gpio.h 文件，重新输入如下内容：

示例代码 17.3.4.1 bsp_gpio.h 文件代码

```
1  #ifndef _BSP_GPIO_H
2  #define _BSP_GPIO_H
3  #include "imx6ul.h"
4  /***************************************************************
5  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6  文件名      : bsp_gpio.h
7  作者        : 左忠凯
8  版本        : V1.0
9  描述        : GPIO 操作文件头文件。
10 其他        : 无
11 论坛        : www.openedv.com
12 日志        : 初版 V1.0 2019/1/4 左忠凯创建
13               V2.0 2019/1/4 左忠凯修改
14               添加 GPIO 中断相关定义
15
16 ***************************************************************/
17
18 /*
19  * 枚举类型和结构体定义
20  */
21 typedef enum _gpio_pin_direction
```

```
22 {
23     kGPIO_DigitalInput = 0U,         /* 输入 */
24     kGPIO_DigitalOutput = 1U,        /* 输出 */
25 } gpio_pin_direction_t;
26 
27 /*
28  * GPIO 中断触发类型枚举
29  */
30 typedef enum _gpio_interrupt_mode
31 {
32     kGPIO_NoIntmode = 0U,                /* 无中断功能          */
33     kGPIO_IntLowLevel = 1U,              /* 低电平触发          */
34     kGPIO_IntHighLevel = 2U,             /* 高电平触发          */
35     kGPIO_IntRisingEdge = 3U,            /* 上升沿触发          */
36     kGPIO_IntFallingEdge = 4U,           /* 下降沿触发          */
37     kGPIO_IntRisingOrFallingEdge = 5U,   /* 上升沿和下降沿都触发 */
38 } gpio_interrupt_mode_t;
39 
40 /*
41  * GPIO 配置结构体
42  */
43 typedef struct _gpio_pin_config
44 {
45     gpio_pin_direction_t direction;      /* GPIO 方向:输入还是输出 */
46     uint8_t outputLogic;                 /* 如果是输出的话，默认输出电平 */
47  gpio_interrupt_mode_t interruptMode;    /* 中断方式 */
48 } gpio_pin_config_t;
49 
50 
51 /* 函数声明 */
52 void gpio_init(GPIO_Type *base, int pin, gpio_pin_config_t *config);
53 int gpio_pinread(GPIO_Type *base, int pin);
54 void gpio_pinwrite(GPIO_Type *base, int pin, int value);
55 void gpio_intconfig(GPIO_Type* base, unsigned int pin,
                        gpio_interrupt_mode_t pinInterruptMode);
56 void gpio_enableint(GPIO_Type* base, unsigned int pin);
57 void gpio_disableint(GPIO_Type* base, unsigned int pin);
58 void gpio_clearintflags(GPIO_Type* base, unsigned int pin);
59 
60 #endif
```

相比前面试验的 bsp_gpio.h 文件，“示例代码 17.3.3.2”中添加了一个新枚举类型：gpio_interrupt_mode_t，枚举出了 GPIO 所有的中断触发类型。还修改了结构体 gpio_pin_config_t，

在，在里面加入了 interruptMode 成员变量。最后就是添加了一些跟中断有关的函数声明，bsp_gpio.h 文件的内容总体还是比较简单的。

打开 bsp_gpio.c 文件，重新输入如下代码：

示例代码 17.3.4.2 bsp_gpio.c 文件代码

```
1  #include "bsp_gpio.h"
2  /***************************************************************
3  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
4  文件名     : bsp_gpio.c
5  作者       : 左忠凯
6  版本       : V1.0
7  描述       : GPIO 操作文件。
8  其他       : 无
9  论坛       : www.openedv.com
10 日志       : 初版 V1.0 2019/1/4 左忠凯创建
11              V2.0 2019/1/4 左忠凯修改:
12              修改 gpio_init()函数，支持中断配置.
13              添加 gpio_intconfig()函数，初始化中断
14              添加 gpio_enableint()函数，使能中断
15              添加 gpio_clearintflags()函数，清除中断标志位
16 
17 ***************************************************************/
18 
19 /*
20  * @description      : GPIO 初始化。
21  * @param - base     : 要初始化的 GPIO 组。
22  * @param - pin      : 要初始化 GPIO 在组内的编号。
23  * @param - config   : GPIO 配置结构体。
24  * @return           : 无
25  */
26 void gpio_init(GPIO_Type *base, int pin, gpio_pin_config_t *config)
27 {
28     base->IMR &= ~(1U << pin);
29 
30     if(config->direction == kGPIO_DigitalInput)  /* GPIO 作为输入 */
31     {
32         base->GDIR &= ~( 1 << pin);
33     }
34     else                                         /* 输出 */
35     {
36         base->GDIR |= 1 << pin;
37         gpio_pinwrite(base,pin, config->outputLogic);/* 设置默认电平 */
38     }
39     gpio_intconfig(base, pin, config->interruptMode);/* 中断功能配置 */
```

```
40 }
41 
42 /*
43  * @description   : 读取指定 GPIO 的电平值 。
44  * @param - base  : 要读取的 GPIO 组。
45  * @param - pin   : 要读取的 GPIO 脚号。
46  * @return        : 无
47  */
48  int gpio_pinread(GPIO_Type *base, int pin)
49  {
50      return (((base->DR) >> pin) & 0x1);
51  }
52 
53 /*
54  * @description   : 指定 GPIO 输出高或者低电平 。
55  * @param - base  : 要输出的的 GPIO 组。
56  * @param - pin   : 要输出的 GPIO 脚号。
57  * @param - value : 要输出的电平，1 输出高电平， 0 输出低低电平
58  * @return        : 无
59  */
60 void gpio_pinwrite(GPIO_Type *base, int pin, int value)
61 {
62     if (value == 0U)
63     {
64         base->DR &= ~(1U << pin); /* 输出低电平 */
65     }
66     else
67     {
68         base->DR |= (1U << pin); /* 输出高电平 */
69     }
70 }
71 
72 /*
73  * @description             : 设置 GPIO 的中断配置功能
74  * @param - base            : 要配置的 IO 所在的 GPIO 组。
75  * @param - pin             : 要配置的 GPIO 脚号。
76  * @param - pinInterruptMode: 中断模式，参考 gpio_interrupt_mode_t
77  * @return                  : 无
78  */
79 void gpio_intconfig(GPIO_Type* base, unsigned int pin, 
                        gpio_interrupt_mode_t pin_int_mode)
80 {
81     volatile uint32_t *icr;
```

```
82      uint32_t icrShift;
83
84      icrShift = pin;
85
86      base->EDGE_SEL &= ~(1U << pin);
87
88      if(pin < 16)    /* 低 16 位 */
89      {
90          icr = &(base->ICR1);
91      }
92      else            /* 高 16 位 */
93      {
94          icr = &(base->ICR2);
95          icrShift -= 16;
96      }
97      switch(pin_int_mode)
98      {
99          case(kGPIO_IntLowLevel):
100             *icr &= ~(3U << (2 * icrShift));
101             break;
102         case(kGPIO_IntHighLevel):
103             *icr = (*icr & (~(3U << (2 * icrShift)))) |
                        (1U << (2 * icrShift));
104             break;
105         case(kGPIO_IntRisingEdge):
106             *icr = (*icr & (~(3U << (2 * icrShift)))) |
                        (2U << (2 * icrShift));
107             break;
108         case(kGPIO_IntFallingEdge):
109             *icr |= (3U << (2 * icrShift));
110             break;
111         case(kGPIO_IntRisingOrFallingEdge):
112             base->EDGE_SEL |= (1U << pin);
113             break;
114         default:
115             break;
116     }
117 }
118
119 /*
120  * @description         : 使能 GPIO 的中断功能
121  * @param - base        : 要使能的 IO 所在的 GPIO 组。
122  * @param - pin         : 要使能的 GPIO 在组内的编号。
```

```
123  * @return              : 无
124  */
125 void gpio_enableint(GPIO_Type* base, unsigned int pin)
126 {
127     base->IMR |= (1 << pin);
128 }
129
130 /*
131  * @description         : 禁止 GPIO 的中断功能
132  * @param - base        : 要禁止的 IO 所在的 GPIO 组。
133  * @param - pin         : 要禁止的 GPIO 在组内的编号。
134  * @return              : 无
135  */
136 void gpio_disableint(GPIO_Type* base, unsigned int pin)
137 {
138     base->IMR &= ~(1 << pin);
139 }
140
141 /*
142  * @description         : 清除中断标志位(写 1 清除)
143  * @param - base        : 要清除的 IO 所在的 GPIO 组。
144  * @param - pin         : 要清除的 GPIO 掩码。
145  * @return              : 无
146  */
147 void gpio_clearintflags(GPIO_Type* base, unsigned int pin)
148 {
149     base->ISR |= (1 << pin);
150 }
```

在 bsp_gpio.c 文件中首先修改了 gpio_init 函数，在此函数里面添加了中断配置代码。另外也新增加了 4 个函数，如下：

gpio_intconfig：配置 GPIO 的中断功能。

gpio_enableint：GPIO 中断使能函数。

gpio_disableint：GPIO 中断禁止函数。

gpio_clearintflags：GPIO 中断标志位清除函数。

bsp_gpio.c 文件重点就是增加了一些跟 GPIO 中断有关的函数，都比较简单。

17.3.5 按键中断驱动文件编写

本例程的目的是以中断的方式编写 KEY 按键驱动，当按下 KEY 以后触发 GPIO 中断，然后在中断服务函数里面控制蜂鸣器的开关。所以接下来就是要编写按键 KEY 对应的 UART1_CTS 这个 IO 的中断驱动，在 bsp 文件夹里面新建名为“exit”的文件夹，然后在 bsp/exit 里面新建 bsp_exit.c 和 bsp_exit.h 两个文件。在 bsp_exit.h 文件中输入如下代码：

示例代码 17.3.5.1 bsp_exit.h 文件代码

```
1  #ifndef _BSP_EXIT_H
```

```
2  #define _BSP_EXIT_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_exit.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : 外部中断驱动头文件。
9  其他       : 配置按键对应的 GPIP 为中断模式
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/4 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 函数声明 */
16 void exit_init(void);                   /* 中断初始化    */
17 void gpio1_io18_irqhandler(void);    /* 中断处理函数 */
18 
19 #endif
```

bsp_exit.h 就是函数声明，很简单。接下来在 bsp_exit.c 里面输入如下内容：

示例代码 17.3.5.2 bsp_exit.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名     : bsp_exit.c
作者       : 左忠凯
版本       : V1.0
描述       : 外部中断驱动。
其他       : 配置按键对应的 GPIP 为中断模式
论坛       : www.openedv.com
日志       : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_exit.h"
2  #include "bsp_gpio.h"
3  #include "bsp_int.h"
4  #include "bsp_delay.h"
5  #include "bsp_beep.h"
6  
7  /*
8   * @description    : 初始化外部中断
9   * @param          : 无
10  * @return         : 无
11  */
12 void exit_init(void)
13 {
```

```
14      gpio_pin_config_t key_config;
15 
16      /* 1、设置 IO 复用 */
17      IOMUXC_SetPinMux(IOMUXC_UART1_CTS_B_GPIO1_IO18,0);
18      IOMUXC_SetPinConfig(IOMUXC_UART1_CTS_B_GPIO1_IO18,0xF080);
19 
20      /* 2、初始化 GPIO 为中断模式 */
21      key_config.direction = kGPIO_DigitalInput;
22      key_config.interruptMode = kGPIO_IntFallingEdge;
23      key_config.outputLogic = 1;
24      gpio_init(GPIO1, 18, &key_config);
25      /* 3、使能 GIC 中断、注册中断服务函数、使能 GPIO 中断 */
26      GIC_EnableIRQ(GPIO1_Combined_16_31_IRQn);
27      system_register_irqhandler(GPIO1_Combined_16_31_IRQn,
                                (system_irq_handler_t)gpio1_io18_irqhandler,
                                NULL);
28      gpio_enableint(GPIO1, 18);
29 }
30 
31 /*
32  * @description    : GPIO1_IO18 最终的中断处理函数
33  * @param          : 无
34  * @return         : 无
35  */
36 void gpio1_io18_irqhandler(void)
37 {
38      static unsigned char state = 0;
39 
40      /*
41      *采用延时消抖，中断服务函数中禁止使用延时函数！因为中断服务需要
42      *快进快出！！这里为了演示所以采用了延时函数进行消抖，后面我们会讲解
43      *定时器中断消抖法！！！
44      */
45 
46      delay(10);
47      if(gpio_pinread(GPIO1, 18) == 0)    /* 按键按下了 */
48      {
49          state = !state;
50          beep_switch(state);
51      }
52 
53      gpio_clearintflags(GPIO1, 18);     /* 清除中断标志位 */
54 }
```

bsp_exit.c 文件只有两个函数 exit_init 和 gpio1_io18_irqhandler，exit_init 是中断初始化函数。第 14~24 行都是初始化 KEY 所使用的 UART1_CTS 这个 IO，设置其复用为 GPIO1_IO18，然后配置 GPIO1_IO18 为下降沿触发中断。重点是第 26~28 行，在 26 行调用函数 GIC_EnableIRQ 来使能 GPIO_IO18 所对应的中断总开关，I.MX6U 中 GPIO1_IO16~IO31 这 16 个 IO 共用 ID99。27 行调用函数 system_register_irqhandler 注册 ID99 所对应的中断处理函数，GPIO1_IO16~IO31 这 16 个 IO 共用一个中断处理函数，至于具体是哪个 IO 引起的中断，那就需要在中断处理函数中判断了。28 行通过函数 gpio_enableint 使能 GPIO1_IO18 这个 IO 对应的中断。

函数 gpio1_io18_irqhandler 就是 27 行注册的中断处理函数，也就是我们学习 STM32 的时候某个 GPIO 对应的中断服务函数。在此函数里面编写中断处理代码，第 50 行就是蜂鸣器开关控制代码，也就是我们本试验的目的。当中断处理完成以后肯定要清除中断标志位，第 53 行调用函数 gpio_clearintflags 来清除 GPIO1_IO18 的中断标志位。

17.3.6 编写 main.c 文件

在 main.c 中输入如下代码：

示例代码 17.3.6.1 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 9 系统中断实验
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_exit.h"
8  
9  /*
10  * @description    : main 函数
11  * @param          : 无
12  * @return         : 无
13  */
14 int main(void)
15 {
16     unsigned char state = OFF;
17 
18     int_init();          /* 初始化中断(一定要最先调用！) */
```

```
19      imx6u_clkinit();        /* 初始化系统时钟           */
20      clk_enable();           /* 使能所有的时钟           */
21      led_init();             /* 初始化 led               */
22      beep_init();            /* 初始化 beep              */
23      key_init();             /* 初始化 key               */
24      exit_init();            /* 初始化按键中断           */
25
26      while(1)
27      {
28          state = !state;
29          led_switch(LED0, state);
30          delay(500);
31      }
32
33      return 0;
34 }
```

main.c 很简单，重点是第 18 行调用函数 int_init 来初始化中断系统，第 24 行调用函数 exit_init 来初始化按键 KEY 对应的 GPIO 中断。

17.4 编译下载验证

17.4.1 编写 Makefile 和链接脚本

在第十六章实验的 Makefile 基础上修改变量 TARGET 为 int，在变量 INCDIRS 和 SRCDIRS 中追加“bsp/exit”和 bsp/int，修改完成以后如下所示：

示例代码 17.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE      ?= arm-linux-gnueabihf-
2  TARGET             ?= int
3
4  /* 省略掉其它代码...... */
5
6  INCDIRS      := imx6ul \
7                  bsp/clk \
8                  bsp/led \
9                  bsp/delay  \
10                 bsp/beep \
11                 bsp/gpio \
12                 bsp/key \
13                 bsp/exit \
14                 bsp/int
15
16 SRCDIRS      := project \
17                 bsp/clk \
18                 bsp/led \
```

```
19                bsp/delay \
20                bsp/beep \
21                bsp/gpio \
22                bsp/key \
23                bsp/exit \
24                bsp/int
25
26 /* 省略掉其它代码...... */
27
28 clean:
29  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“int”，也就是目标名称为“int”。

第 13、14 行在变量 INCDIRS 中添加 GPIO 中断和通用中断驱动头文件(.h)路径。

第 23、24 行在变量 SRCDIRS 中添加 GPIO 中断和通用中断驱动文件(.c)路径。

链接脚本保持不变。

17.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 int.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload            //给予 imxdownload 可执行权限，一次即可
./imxdownload int.bin /dev/sdd   //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。本试验效果其实和试验“8_key”一样，按下 KEY 就会打开蜂鸣器，再次按下就会关闭蜂鸣器。LED0 会不断闪烁，周期大约 500ms。

第十八章 EPIT 定时器试验

定时器是最常用的外设，常常需要使用定时器来完成精准的定时功能，I.MX6U 提供了多种硬件定时器，有些定时器功能非常强大。本章我们从最基本的 EPIT 定时器开始，学习如何配置 EPIT 定时器，使其按照给定的时间，周期性的产生定时器中断，在定时器中断里面我们可以做其它的处理，比如翻转 LED 灯。

18.1 EPIT 定时器简介

EPIT 的全称是：Enhanced Periodic Interrupt Timer，直译过来就是增强的周期中断定时器，它主要是完成周期性中断定时的。学过 STM32 的话应该知道，STM32 里面的定时器还有很多其它的功能，比如输入捕获、PWM 输出等等。但是 I.MX6U 的 EPIT 定时器只是完成周期性中断定时的，仅此一项功能！至于输入捕获、PWM 输出等这些功能，I.MX6U 有其它的外设来完成。

EPIT 是一个 32 位定时器，在处理器几乎不用介入的情况下提供精准的定时中断，软件使能以后 EPIT 就会开始运行，EPIT 定时器有如下特点：

、时钟源可选的 32 位向下计数器。

、12 位的分频值。

、当计数值和比较值相等的时候产生中断。

EPIT 定时器结构如图 18.1.1 所示：

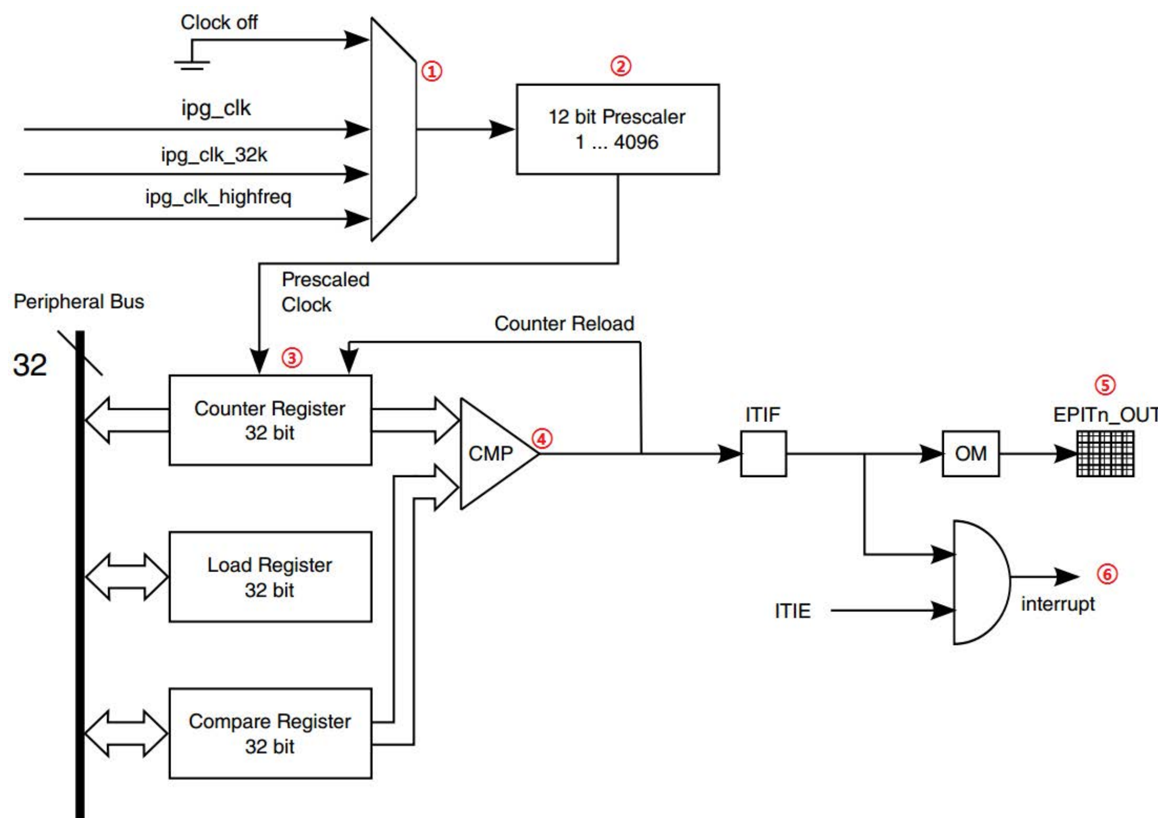


图 18.1.1 EPIT 定时器框图

图 18.1.1 中各部分的功能如下：

、这是个多路选择器，用来选择 EPIT 定时器的时钟源，EPIT 共有 3 个时钟源可选择，ipg_clk、ipg_clk_32k 和 ipg_clk_highfreq。

、这是一个 12 位的分频器，负责对时钟源进行分频，12 位对应的值是 0~4095，对应着 1~4096 分频。

、经过分频的时钟进入到 EPIT 内部，在 EPIT 内部有三个重要的寄存器：计数寄存器(EPIT_CNR)、加载寄存器(EPIT_LR)和比较寄存器(EPIT_CMPR)，这三个寄存器都是 32 位的。EPIT 是一个向下计数器，也就是说给它一个初值，它就会从这个给定的初值开始递减，直到减为 0，计数寄存器里面保存的就是当前的计数值。如果 EPIT 工作在 set-and-forget 模式下，当计

数寄存器里面的值减少到 0，EPIT 就会重新从加载寄存器读取数值到计数寄存器里面，重新开始向下计数。比较寄存器里面保存的数值用于和计数寄存器里面的计数值比较，如果相等的话就会产生一个比较事件。

、EPIT 可以设置引脚输出，如果设置了的话就会通过指定的引脚输出信号。

、产生比较中断，也就是定时中断。

EPIT 定时器有两种工作模式：set-and-forget 和 free-running，这两个工作模式的区别如下：

set-and-forget 模式：EPITx_CR(x=1，2)寄存器的 RLD 位置 1 的时候 EPIT 工作在此模式下，在此模式下 EPIT 的计数器从加载寄存器 EPITx_LR 中获取初始值，不能直接向计数器寄存器写入数据。不管什么时候，只要计数器计数到 0，那么就会从加载寄存器 EPITx_LR 中重新加载数据到计数器中，周而复始。

free-running 模式：EPITx_CR 寄存器的 RLD 位清零的时候 EPIT 工作在此模式下，当计数器计数到 0 以后会重新从 0XFFFFFFFF 开始计数，并不是从加载寄存器 EPITx_LR 中获取数据。

接下来看一下 EPIT 重要的几个寄存器，第一个就是 EPIT 的配置寄存器 EPITx_CR，此寄存器的结构如图 18.1.2 所示：

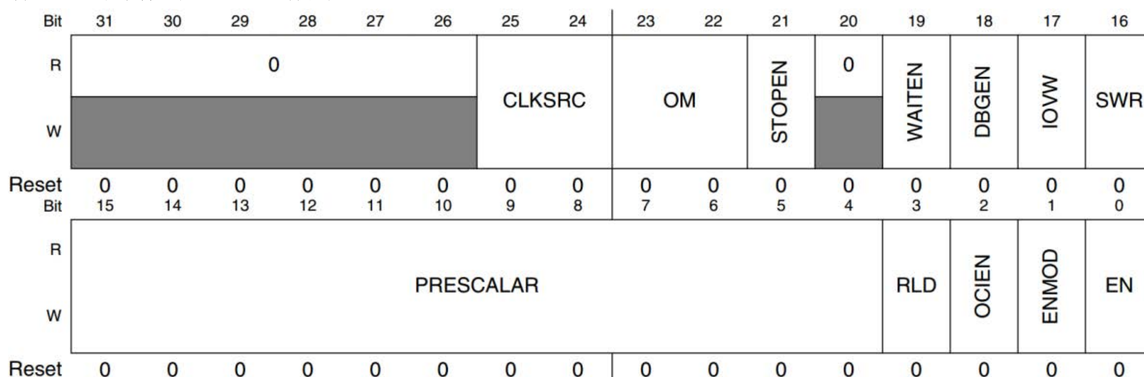


图 18.1.2 EPITx_CR 寄存器结构图

寄存器 EPITx_CR 我们用到的重要位如下：

CLKSRC(bit25:24)：EPIT 时钟源选择位，为 0 的时候关闭时钟源，1 的时候选择选择 Peripheral 时钟(ipg_clk)，为 2 的时候选择 High-frequency 参考时钟(ipg_clk_highfreq)，为 3 的时候选择 Low-frequency 参考时钟(ipg_clk_32k)。在本例程中，我们设置为 1，也就是选择 ipg_clk 作为 EPIT 的时钟源，ipg_clk=66MHz。

PRESCALAR(bit15:4)：EPIT 时钟源分频值，可设置范围 0~4095，分别对应 1~4096 分频。

RLD(bit3)：EPIT 工作模式，为 0 的时候工作在 free-running 模式，为 1 的时候工作在 set-and-forget 模式。本章例程设置为 1，也就是工作在 set-and-forget 模式。

OCIEN(bit2)：比较中断使能位，为 0 的时候关闭比较中断，为 1 的时候使能比较中断，本章试验要使能比较中断。

ENMOD(bit1)：设置计数器初始值，为 0 时计数器初始值等于上次关闭 EPIT 定时器以后计数器里面的值，为 1 的时候来源于加载寄存器。

EN(bit0)：EPIT 使能位，为 0 的时候关闭 EPIT，为 1 的时候使能 EPIT。

寄存器 EPITx_SR 结构体如图 18.1.3 所示：

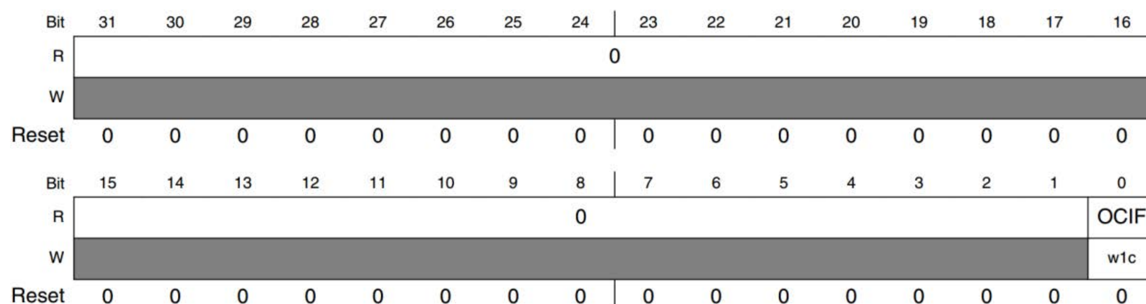


图 18.1.3 EPITx_SR 寄存器结构图

寄存器 EPITx_SR 只有一个位有效，那就是 OCIF(bit0)，这个位是比较中断标志位，为 0 的时候表示没有比较事件发生，为 1 的时候表示有比较事件发生。当比较中断发生以后需要手动清除此位，此位是写 1 清零的。

寄存器 EPITx_LR、EPITx_CMPR 和 EPITx_CNR 分别为加载寄存器、比较寄存器和计数寄存器，这三个寄存器都是用来存放数据的，很简单。

关于 EPIT 的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 1174 页的 24.6 小节。本章我们使用 EPIT 产生定时中断，然后在中断服务函数里面翻转 LED0，接下来以 EPIT1 为例，讲解需要哪些步骤来实现这个功能。EPIT 的配置步骤如下：

1、设置 EPIT1 的时钟源

设置寄存器 EPIT1_CR 寄存器的 CLKSRC(bit25:24)位，选择 EPIT1 的时钟源。

2、设置分频值

设置寄存器 EPIT1_CR 寄存器的 PRESCALAR(bit15:4)位，设置分频值。

3、设置工作模式

设置寄存器 EPIT1_CR 的 RLD(bit3)位，设置 EPTI1 的工作模式。

4、设置计数器的初始值来源

设置寄存器 EPIT1_CR 的 ENMOD(bit1)位，设置计数器的初始值来源。

5、使能比较中断

我们要使用到比较中断，因此需要设置寄存器 EPIT1_CR 的 OCIEN(bit2)位，使能比较中断。

6、设置加载值和比较值

设置寄存器 EPIT1_LR 中的加载值和寄存器 EPIT1_CMPR 中的比较值，通过这两个寄存器就可以决定定时器的中断周期。

7、EPIT1 中断设置和中断服务函数编写

使能 GIC 中对应的 EPIT1 中断，注册中断服务函数，如果需要的话还可以设置中断优先级。最后编写中断服务函数。

8、使能 EPIT1 定时器

配置好 EPIT1 以后就可以使能 EPIT1 了，通过寄存器 EPIT1_CR 的 EN(bit0)位来设置。

通过以上几步我们就配置好 EPIT 了，通过 EPIT 的比较中断来实现 LED0 的翻转。

18.2 硬件原理分析

本试验用到的资源如下：

①、LED0。

②、 定时器 EPTI1。

本实验通过 EPTI1 的中断来控制 LED0 的亮灭，LED0 的硬件原理前面已经介绍过了。

18.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 10_epit_timer。

本章实验在上一章例程的基础上完成，更改工程名字为“epit_timer”，然后在 bsp 文件夹下创建名为“epittimer”的文件夹，然后在 bsp/epittimer 中新建 bsp_epittimer.c 和 bsp_epittimer.h 这两个文件。在 bsp_epittimer.h 中输入如下内容：

示例代码 18.3.1 bsp_epittimer.h 文件代码

```
1  #ifndef _BSP_EPITTIMER_H
2  #define _BSP_EPITTIMER_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_epittimer.h
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : EPIT 定时器驱动头文件。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/1/5 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14
15 /* 函数声明 */
16 void epit1_init(unsigned int frac, unsigned int value);
17 void epit1_irqhandler(void);
18
19 #endif
```

bsp_epittimer.h 文件很简单，就是一些函数声明。然后在 bsp_epittimer.c 中输入如下内容：

示例代码 18.3.2 bsp_epittimer.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_epittimer.c
作者      : 左忠凯
版本      : V1.0
描述      : EPIT 定时器驱动文件。
其他      : 配置 EPIT 定时器，实现 EPIT 定时器中断处理函数
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/5 左忠凯创建
```

```
***************************************************************/
1  #include "bsp_epittimer.h"
2  #include "bsp_int.h"
3  #include "bsp_led.h"
4
5  /*
6   * @description    : 初始化 EPIT 定时器.
7   *                   EPIT 定时器是 32 位向下计数器,时钟源使用 ipg=66Mhz
8   * @param – frac   : 分频值，范围为 0~4095，分别对应 1~4096 分频。
9   * @param - value  : 倒计数值。
10  * @return         : 无
11  */
12 void epit1_init(unsigned int frac, unsigned int value)
13 {
14     if(frac > 0XFFF)
15         frac = 0XFFF;
16     EPIT1->CR = 0;  /* 先清零 CR 寄存器 */
17
18     /*
19      * CR 寄存器:
20      * bit25:24 01 时钟源选择 Peripheral clock=66MHz
21      * bit15:4  frac 分频值
22      * bit3: 1  当计数器到 0 的话从 LR 重新加载数值
23      * bit2: 1  比较中断使能
24      * bit1:    1  初始计数值来源于 LR 寄存器值
25      * bit0:    0  先关闭 EPIT1
26      */
27     EPIT1->CR = (1<<24 | frac << 4 | 1<<3 | 1<<2 | 1<<1);
28     EPIT1->LR = value;   /* 加载寄存器值 */
29     EPIT1->CMPR = 0;     /* 比较寄存器值 */
30
31     /* 使能 GIC 中对应的中断 */
32     GIC_EnableIRQ(EPIT1_IRQn);
33
34     /* 注册中断服务函数     */
35     system_register_irqhandler(EPIT1_IRQn,
                                (system_irq_handler_t)epit1_irqhandler,
                                NULL);
36     EPIT1->CR |= 1<<0;  /* 使能 EPIT1 */
37 }
38
39 /*
40  * @description    : EPIT 中断处理函数
```

```
41  * @param          : 无
42  * @return         : 无
43  */
44 void epit1_irqhandler(void)
45 {
46     static unsigned char state = 0;
47     state = !state;
48     if(EPIT1->SR & (1<<0))          /* 判断比较事件发生        */
49     {
50         led_switch(LED0, state);    /* 定时器周期到，反转 LED   */
51     }
52     EPIT1->SR |= 1<<0;              /* 清除中断标志位          */
53 }
```

bsp_epittimer.c 里面有两个函数 epit1_init 和 epit1_irqhandler，分别是 EPIT1 初始化函数和 EPIT1 中断处理函数。epit1_init 有两个参数 frac 和 value，其中 frac 是分频值，value 是加载值。在第 29 行设置比较寄存器为 0，也就是当计数器倒计数到 0 以后就会触发比较中断，因此分频值 frac 和 value 就可以决定中断频率，计算公式如下：

$$Tout = ((frac + 1) * value) / Tclk;$$

其中：

Tclk：EPIT1 的输入时钟频率(单位 Hz)。

Tout：EPIT1 的溢出时间(单位 S)。

第 38 行设置了 EPIT1 工作模式为 set-and-forget，并且时钟源为 ipg_clk=66MHz。假如我们现在要设置 EPIT1 中断周期为 500ms，可以设置分频值为 0，也就是 1 分频，这样进入 EPIT1 的时钟就是 66MHz。如果要实现 500ms 的中断周期，EPIT1 的加载寄存器就应该为 66000000/2=33000000。

函数 epit1_irqhandler 是 EPIT1 的中断处理函数，此函数先读取 EPIT1_SR 寄存器，判断当前的中断是否为比较事件，如果是的话就翻转 LED 灯。最后在退出中断处理函数的时候需要清除中断标志位。

最后就是 mian.c 文件了，在 mian.c 里面输入如下内容：

示例代码 18.3.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 10 EPIT 定时器实验
其他      : 本实验主要学习使用 I.MX6UL 自带的 EPIT 定时器，学习如何使用
            EPIT 定时器来实现定时功能，巩固 Cortex-A 的中断知识。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/4 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
```

```
3 #include "bsp_led.h"
4 #include "bsp_beep.h"
5 #include "bsp_key.h"
6 #include "bsp_int.h"
7 #include "bsp_epittimer.h"
8
9 /*
10  * @description  : main 函数
11  * @param        : 无
12  * @return       : 无
13  */
14 int main(void)
15 {
16     int_init();                  /* 初始化中断(一定要最先调用！) */
17     imx6u_clkinit();             /* 初始化系统时钟               */
18     clk_enable();                /* 使能所有的时钟               */
19     led_init();                  /* 初始化 led                   */
20     beep_init();                 /* 初始化 beep                  */
21     key_init();                  /* 初始化 key                   */
22     epit1_init(0, 66000000/2);   /* 初始化 EPIT1 定时器，1 分频
23                                   * 计数值为:66000000/2，也就是
24                                   * 定时周期为 500ms。
25                                   */
26     while(1)
27     {
28         delay(500);
29     }
30
31     return 0;
32 }
```

main.c 里面就一个 main 函数，第 22 行调用函数 epit1_init 来初始化 EPIT1，分频值为 0，也就是 1 分频，加载寄存器值为 66000000/2=33000000，EPTI1 定时器中断周期为 500ms。第 26~29 行的 while 循环里面就只有一个延时函数，没有做其他处理，延时函数都可以取掉。

18.4 编译下载验证

18.4.1 编写 Makefile 和链接脚本

修改 Makfile 中的 TARGET 为 epit，在 INCDIRS 和 SRCDIRS 中加入“bsp/epittimer”，修改后的 Makefile 如下：

示例代码 18.4.1.1 Makefile 文件代码

```
1 CROSS_COMPILE   ?= arm-linux-gnueabihf-
2 TARGET          ?= epit
3
```

```
4  /* 省略掉其它代码...... */
5
6  INCDIRS         :=  imx6ul \
7                      bsp/clk \
8                      bsp/led \
9                      bsp/delay  \
10                     bsp/beep \
11                     bsp/gpio \
12                     bsp/key \
13                     bsp/exit \
14                     bsp/int \
15                     bsp/epittimer
16
17 SRCDIRS          := project \
18                     bsp/clk \
19                     bsp/led \
20                     bsp/delay \
21                     bsp/beep \
22                     bsp/gpio \
23                     bsp/key \
24                     bsp/exit \
25                     bsp/int \
26                     bsp/epittimer
27
28  /* 省略掉其他代码...... */
29
30 clean:
31  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“epit”，也就是目标名称为“epit”。

第 15 行在变量 INCDIRS 中添加 EPIT1 驱动头文件(.h)路径。

第 26 行在变量 SRCDIRS 中添加 EPIT1 驱动文件(.c)路径。

链接脚本保持不变。

18.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 epit.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                 //给予 imxdownload 可执行权限，一次即可
./imxdownload epit.bin /dev/sdd       //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。程序运行正常的话 LED0 会以 500ms 为周期不断的亮、灭闪烁。

第十九章 定时器按键消抖实验

在第十五章和第十七章实验中都用到了按键，用到按键就要处理因为机械结构带来的按键抖动问题，也就是按键消抖。前面的实验中都是直接使用了延时函数来实现消抖，因为简单，但是直接用延时函数来实现消抖会浪费 CPU 性能，因为在延时函数里面 CPU 什么都做不了。如果按键使用中断的话更不能在中断里面使用延时函数，因为中断服务函数要快进快出！本章我们学习如何使用定时器来实现按键消抖，使用定时器既可以实现按键消抖，而且也不会浪费 CPU 性能，这个也是 Linux 驱动里面按键消抖的做法。

19.1 定时器按键消抖简介

按键消抖的原理在第十五章已经详细的讲解了，其实就是在按键按下以后延时一段时间再去读取按键值，如果此时按键值还有效那就表示这是一次有效的按键，中间的延时就是消抖的。但是这有一个缺点，就是延时函数会浪费 CPU 性能，因为延时函数就是空跑。如果按键是用中断方式实现的，那就更不能在中断服务函数里面使用延时函数，因为中断服务函数最基本的要求就是快进快出！上一章我们学习了 EPIT 定时器，定时器设置好定时时间，然后 CPU 就可以做其他事情去了，定时时间到了以后就会触发中断，然后在中断中做相应的处理即可。因此，我们可以借助定时器来实现消抖，按键采用中断驱动方式，当按键按下以后触发按键中断，在按键中断中开启一个定时器，定时周期为 10ms，当定时时间到了以后就会触发定时器中断，最后在定时器中断处理函数中读取按键的值，如果按键值还是按下状态那就表示这是一次有效的按键。定时器按键消抖如图 19.1.1 所示：

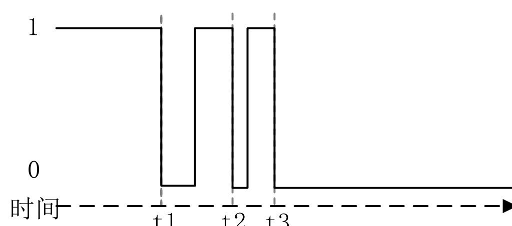


图 19.1.1 定时器消抖示意图

在图 19.1.1 中 t1~t3 这一段时间就是按键抖动，是需要消除的。设置按键为下降沿触发，因此会在 t1、t2 和 t3 这三个时刻会触发按键中断，每次进入中断处理函数都会重新开器定时器中断，所以会在 t1、t2 和 t3 这三个时刻开器定时器中断。但是 t1~t2 和 t2~t3 这两个时间段是小于我们设置的定时器中断周期(也就是消抖时间，比如 10ms)，所以虽然 t1 开启了定时器，但是定时器定时时间还没到呢 t2 时刻就重置了定时器，最终只有 t3 时刻开启的定时器能完整的完成整个定时周期并触发中断，我们就可以在中断处理函数里面做按键处理了，这就是定时器实现按键防抖的原理，Linux 里面的按键驱动用的就是这个原理！

关于定时器按键消抖的原理就介绍到这里，接下来讲解如何使用 EPIT1 来配合按键 KEY 来实现具体的消抖，步骤如下：

1、配置按键 IO 中断

配置按键所使用的 IO，因为要使用到中断驱动按键，所以要配置 IO 的中断模式。

2、初始化消抖用的定时器

上面已经讲的很清楚了，消抖要用定时器来完成，所以需要初始化一个定时器，这里使用上一章讲解的 EPIT1 定时器，也算是对 EPIT1 定时器的一次巩固。定时器的定时周期为 10ms，也可根据实际情况调整定时周期。

3、编写中断处理函数

需要编写两个中断处理函数：按键对应的 GPIO 中断处理函数和 EPIT1 定时器的中断处理函数。在按键的中断处理函数中主要用于开启 EPIT1 定时器，EPIT1 的中断处理函数才是重点，按键要做的具体任务都是在定时器 EPIT1 的中断处理函数中完成的，比如控制蜂鸣器打开或关闭。

19.2 硬件原理分析

本试验用到的资源如下：

、一个 LED 灯 LED0。

、定时器 EPTI1。

、一个按键 KEY。

、一个蜂鸣器。

本试验效果和第十五章的试验效果一样，按下 KEY 会打开蜂鸣器，再次按下 KEY 就会关闭蜂鸣器。LED0 作为系统提示灯不断的闪烁。

19.3 试验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 11_key_filter。

本章实验在上一章例程的基础上完成，更改工程名字为“key_filter”，然后在 bsp 文件夹下创建名为“keyfilter”的文件夹，然后在 bsp/keyfilter 中新建 bsp_keyfilter.c 和 bsp_keyfilter.h 这两个文件。在 bsp_keyfilter.h 中输入如下内容：

示例代码 19.3.1 bsp_keyfilter.h 文件代码

```
1  #ifndef _BSP_KEYFILTER_H
2  #define _BSP_KEYFILTER_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_keyfilter.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : 定时器按键消抖驱动头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/5 左忠凯创建
12 ***************************************************************/
13 
14 /* 函数声明 */
15 void filterkey_init(void);
16 void filtertimer_init(unsigned int value);
17 void filtertimer_stop(void);
18 void filtertimer_restart(unsigned int value);
19 void filtertimer_irqhandler(void);
20 void gpio1_16_31_irqhandler(void);
21 
22 #endif
```

bsp_keyfilter.h 文件很简单，只是函数声明。在 bsp_keyfilter.c 中输入如下内容：

示例代码 19.3.2 bsp_keyfilter.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : bsp_keyfilter.c
```

```
作者      : 左忠凯
版本      : V1.0
描述      : 定时器按键消抖驱动。
其他      : 按键采用中断方式，按下按键触发按键中断，在按键中断里面
            使能定时器定时中断。使用定时器定时中断来完成消抖延时，
            定时器中断周期就是延时时间。如果定时器定时中断触发，
            表示消抖完成(延时周期完成)，即可执行按键处理函数。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/5 左忠凯创建
****************************************************************/
1   #include "bsp_key.h"
2   #include "bsp_gpio.h"
3   #include "bsp_int.h"
4   #include "bsp_beep.h"
5   #include "bsp_keyfilter.h"
6
7   /*
8    * @description   : 按键初始化
9    * @param         : 无
10   * @return        : 无
11   */
12  void filterkey_init(void)
13  {
14      gpio_pin_config_t key_config;
15
16      /* 1、初始化 IO */
17      IOMUXC_SetPinMux(IOMUXC_UART1_CTS_B_GPIO1_IO18, 0);
18      IOMUXC_SetPinConfig(IOMUXC_UART1_CTS_B_GPIO1_IO18, 0xF080);
19
20      /* 2、初始化 GPIO 为中断 */
21      key_config.direction = kGPIO_DigitalInput;
22      key_config.interruptMode = kGPIO_IntFallingEdge;
23      key_config.outputLogic = 1;
24      gpio_init(GPIO1, 18, &key_config);
25
26      /* 3、 使能 GPIO 中断，并且注册中断处理函数        */
27      GIC_EnableIRQ(GPIO1_Combined_16_31_IRQn);
28      system_register_irqhandler(GPIO1_Combined_16_31_IRQn,
                        (system_irq_handler_t)gpio1_16_31_irqhandler,
                        NULL);
29
30      gpio_enableint(GPIO1, 18);      /* 使能 GPIO1_IO18 的中断功能   */
31      filtertimer_init(66000000/100); /* 初始化定时器,10ms            */
```

```
32 }
33
34 /*
35  * @description       : 初始化用于消抖的定时器，默认关闭定时器
36  * @param - value     : 定时器 EPIT 计数值
37  * @return            : 无
38  */
39 void filtertimer_init(unsigned int value)
40 {
41     EPIT1->CR = 0;          /* 先清零            */
42     EPIT1->CR = (1<<24 | 1<<3 | 1<<2 | 1<<1);
43     EPIT1->LR = value;      /* 计数值            */
44     EPIT1->CMPR = 0;        /* 比较寄存器为 0    */
45
46     /* 使能 EPIT1 中断并注册中断处理函数*/
47     GIC_EnableIRQ(EPIT1_IRQn);
48     system_register_irqhandler(EPIT1_IRQn,
                         (system_irq_handler_t)filtertimer_irqhandler,
                         NULL);
49 }
50
51 /*
52  * @description    : 关闭定时器
53  * @param          : 无
54  * @return         : 无
55  */
56 void filtertimer_stop(void)
57 {
58     EPIT1->CR &= ~(1<<0);      /* 关闭定时器    */
59 }
60
61 /*
62  * @description    : 重启定时器
63  * @param – value : 定时器 EPIT 计数值
64  * @return         : 无
65  */
66 void filtertimer_restart(unsigned int value)
67 {
68     EPIT1->CR &= ~(1<<0);      /* 先关闭定时器 */
69     EPIT1->LR = value;         /* 计数值       */
70     EPIT1->CR |= (1<<0);       /* 打开定时器   */
71 }
72
```

```
73  /*
74   * @description   : 定时器中断处理函数
75   * @param         : 无
76   * @return        : 无
77   */
78  void filtertimer_irqhandler(void)
79  {
80      static unsigned char state = OFF;
81  
82      if(EPIT1->SR & (1<<0))                  /* 判断比较事件是否发生  */
83      {
84          filtertimer_stop();                 /* 关闭定时器           */
85          if(gpio_pinread(GPIO1, 18) == 0)    /* KEY0 按下            */
86          {
87              state = !state;
88              beep_switch(state);             /* 反转蜂鸣器           */
89          }
90      }
91      EPIT1->SR |= 1<<0;                      /* 清除中断标志位        */
92  }
93  
94  /*
95   * @description   : GPIO 中断处理函数
96   * @param         : 无
97   * @return        : 无
98   */
99  void gpio1_16_31_irqhandler(void)
100 {
101     filtertimer_restart(66000000/100); /* 开启定时器       */
102     gpio_clearintflags(GPIO1, 18);     /* 清除中断标志位   */
103 }
```

文件 bsp_keyfilter.c 一共有 6 个函数，这 6 个函数其实都很简单。filterkey_init 是本试验的初始化函数，此函数首先初始化了 KEY 所使用的 UART1_CTS 这个 IO，设置这个 IO 的中断模式，并且注册中断处理函数，最后调用函数 filtertimer_init 初始化定时器 EPIT1 定时周期为 10ms。函数 filtertimer_init 是定时器 EPIT1 的初始化函数，内容基本和上一章实验的 EPIT1 初始化函数一样。函数 filtertimer_stop 和 filtertimer_restart 分别是 EPIT1 的关闭和重启函数。filtertimer_irqhandler 是 EPTI1 的中断处理函数，此函数里面就是按键要做的工作，在本例程里面就是开启或者关闭蜂鸣器。函数 gpio1_16_31_irqhandler 是 GPIO1_IO18 的中断处理函数，此函数只有一个工作，那就是重启定时器 EPIT1。

bsp_keyfilter.c 文件内容总体来说并不难，基本就是第十七章和第十八章实验的综合。最后在 mian.c 中输入如下所示代码：

示例代码 19.3.3 main.c 文件代码

```
/***************************************************************
```

```
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名      : mian.c
作者        : 左忠凯
版本        : V1.0
描述        : I.MX6U 开发板裸机实验 11 定时器实现按键消抖实验
其他        : 本实验主要学习如何使用定时器来实现按键消抖，以前的按键
              消抖都是直接使用延时函数来完成的，这种做法效率不高，因为
              延时函数完全是浪费 CPU 资源的。使用按键中断+定时器来实现按键
              驱动效率是最好的，这也是 Linux 驱动所使用的方法！
论坛        : www.openedv.com
日志        : 初版 V1.0 2019/1/5 左忠凯创建
****************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_keyfilter.h"
8
9  /*
10  * @description  : main 函数
11  * @param        : 无
12  * @return       : 无
13  */
14 int main(void)
15 {
16     unsigned char state = OFF;
17
18     int_init();                 /* 初始化中断(一定要最先调用！) */
19     imx6u_clkinit();            /* 初始化系统时钟             */
20     clk_enable();               /* 使能所有的时钟             */
21     led_init();                 /* 初始化 led                 */
22     beep_init();                /* 初始化 beep                */
23     filterkey_init();           /* 带有消抖功能的按键         */
24
25     while(1)
26     {
27         state = !state;
28         led_switch(LED0, state);
29         delay(500);
30     }
31
```

```
32     return 0;
33 }
```

main.c 文件只有一个 main 函数，在第 23 行调用函数 filterkey_init 来初始化带有消抖的按键，最后在 while 循环里面翻转 LED0，周期大约为 500ms。

19.4 编译下载验证

19.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 keyfilter，在 INCDIRS 和 SRCDIRS 中加入“bsp/keyfilter”，修改后的 Makefile 如下：

示例代码 19.4.1 Makefile 代码

```
1  CROSS_COMPILE     ?= arm-linux-gnueabihf-
2  TARGET            ?= keyfilter
3  
4  /* 省略掉其它代码...... */
5  
6  INCDIRS           :=  imx6ul \
7                        bsp/clk \
8                        bsp/led \
9                        bsp/delay  \
10                       bsp/beep \
11                       bsp/gpio \
12                       bsp/key \
13                       bsp/exit \
14                       bsp/int \
15                       bsp/epittimer \
16                       bsp/keyfilter
17 
18 SRCDIRS           :=  project \
19                       bsp/clk \
20                       bsp/led \
21                       bsp/delay \
22                       bsp/beep \
23                       bsp/gpio \
24                       bsp/key \
25                       bsp/exit \
26                       bsp/int \
27                       bsp/epittimer \
28                       bsp/keyfilter
29 
30 /* 省略掉其它代码...... */
31 
32 clean:
```

```
33    rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“keyfilter”，也就是目标名称为“keyfilter”。

第 16 行在变量 INCDIRS 中添加按键消抖驱动头文件(.h)路径。

第 28 行在变量 SRCDIRS 中添加按键消抖驱动文件(.c)路径。

链接脚本保持不变。

19.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 keyfilter.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload keyfilter.bin /dev/sdd     //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。本例程的效果和第十五章一样，按下 KEY 就会控制蜂鸣器的开关，并且 LED0 不断的闪烁，提示系统正在运行。

第二十章 高精度延时实验

延时函数是很常用的 API 函数，在前面的实验中我们使用循环来实现延时函数，但是使用循环来实现的延时函数不准确，误差会很大。虽然使用到延时函数的地方精度要求都不会很严格(要求严格的话就使用硬件定时器了)，但是延时函数肯定是越精确越好，这样延时函数就可以使用在某些对时许要求严格的场合。本章我们就来学习一下如何使用硬件定时器来实现高精度延时。

20.1 高精度延时简介

20.1.1 GPT 定时器简介

学过 STM32 的同学应该知道，在使用 STM32 的时候可以使用 SYSTICK 来实现高精度延时。I.MX6U 没有 SYSTICK 定时器，但是 I.MX6U 有其他定时器啊，比如第十八章讲解的 EPIT 定时器。本章我们使用 I.MX6U 的 GPT 定时器来实现高精度延时，顺便学习一下 GPT 定时器，GPT 定时器全称为 General Purpose Timer。

GPT 定时器是一个 32 位向上定时器(也就是从 0X00000000 开始向上递增计数)，GPT 定时器也可以跟一个值进行比较，当计数器值和这个值相等的话就发生比较事件，产生比较中断。GPT 定时器有一个 12 位的分频器，可以对 GPT 定时器的时钟源进行分频，GPT 定时器特性如下：

①、一个可选时钟源的 32 位向上计数器。

②、两个输入捕获通道，可以设置触发方式。

③、三个输出比较通道，可以设置输出模式。

④、可以生成捕获中断、比较中断和溢出中断。

⑤、计数器可以运行在重新启动(restart)或(自由运行)free-run 模式。

GPT 定时器的可选时钟源如图 20.1.1.1 所示：

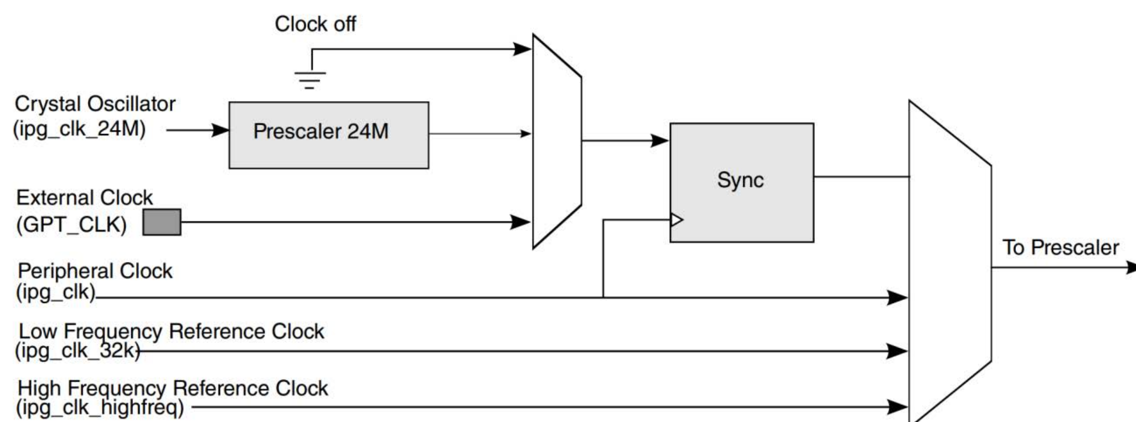


图 20.1.1.1 GPT 时钟源

从图 20.1.1.1 可以看出一共有五个时钟源，分别为：ipg_clk_24M、GPT_CLK(外部时钟)、ipg_clk、ipg_clk_32k 和 ipg_clk_highfreq。本例程选择 ipg_clk 为 GPT 的时钟源，ipg_clk-66MHz。

GPT 定时器结构如图 20.1.1.2 所示：

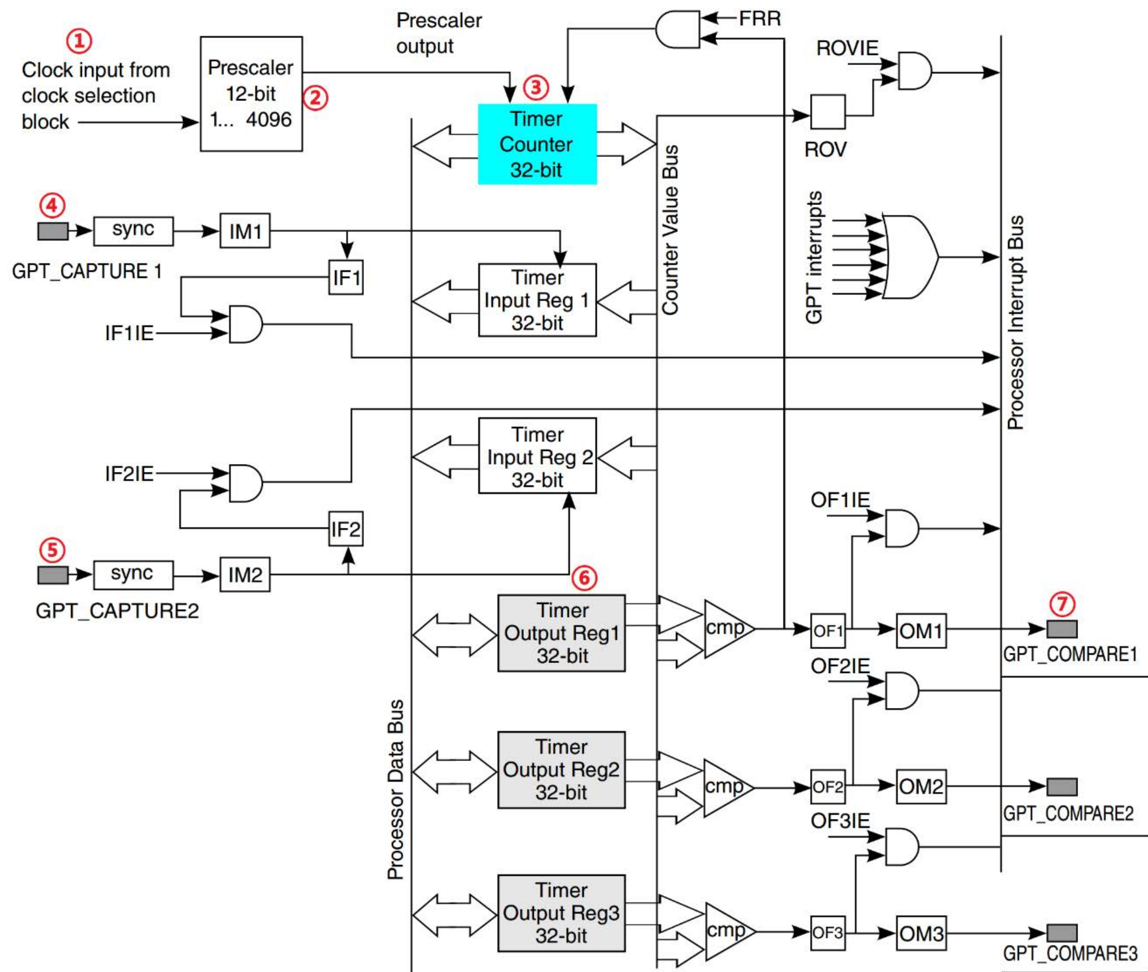


图 20.1.1.2 GPT 定时器结构图

图 20.1.1.2 中各部分意义如下：

①、此部分为 GPT 定时器的时钟源，前面已经说过了，本章例程选择 ipg_clk 作为 GPT 定时器时钟源。

②、此部分为 12 位分频器，对时钟源进行分频处理，可设置 0~4095，分别对应 1~4096 分频。

③、经过分频的时钟源进入到 GPT 定时器内部 32 位计数器。

④和⑤、这两部分是 GPT 的两路输入捕获通道，本章不讲解 GPT 定时器的输入捕获。

⑥、此部分为输出比较寄存器，一共有三路输出比较，因此有三个输出比较寄存器，输出比较寄存器是 32 位的。

⑦、此部分位输出比较中断，三路输出比较中断，当计数器里面的值和输出比较寄存器里面的比较值相等就会触发输出比较中断。

GPT 定时器有两种工作模式：重新启动(restart)模式和自由运行(free-run)模式，这两个工作模式的区别如下：

重新启动(restart)模式：当 GPTx_CR(x=1，2)寄存器的 FRR 位清零的时候 GPT 工作在此模式。在此模式下，当计数值和比较寄存器中的值相等的话计数值就会清零，然后重新从 0X00000000 开始向上计数，只有比较通道 1 才有此模式！向比较通道 1 的比较寄存器写入任何数据都会复位 GPT 计数器。对于其他两路比较通道（通道 2 和 3），当发生比较事件以后不会复位计数器。

自由运行(free-run)模式：当 GPTx_CR(x=1，2)寄存器的 FRR 位置 1 时候 GPT 工作在此模式下，此模式适用于所有三个比较通道，当比较事件发生以后并不会复位计数器，而是继续计数，直到计数值为 0XFFFFFFFF，然后重新回滚到 0X00000000。

接下来看一下 GPT 定时器几个重要的寄存器，第一个就是 GPT 的配置寄存器 GPTx_CR，此寄存器的结构如图 20.1.1.3 所示：

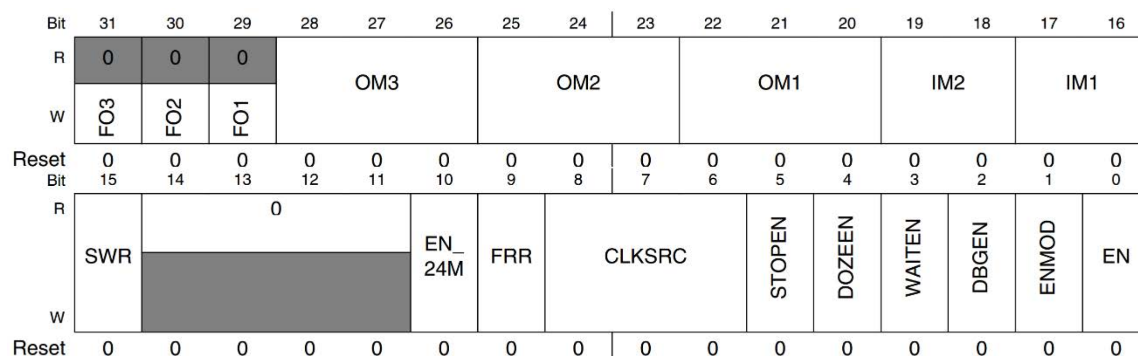


图 20.1.1.3 寄存器 GPTx_CR

寄存器 GPTx_CR 我们用到的重要位如下：

SWR(bit15)：复位 GPT 定时器，向此位写 1 就可以复位 GPT 定时器，当 GPT 复位完成以后此为会自动清零。

FRR(bit9)：运行模式选择，当此位为 0 的时候比较通道 1 工作在重新启动(restart)模式。当此位为 1 的时候所有的三个比较通道均工作在自由运行模式(free-run)。

CLKSRC(bit8:6)：GPT 定时器时钟源选择位，为 0 的时候关闭时钟源；为 1 的时候选择 ipg_clk 作为时钟源；为 2 的时候选择 ipg_clk_highfreq 为时钟源；为 3 的时候选择外部时钟为时钟源；为 4 的时候选择 ipg_clk_32k 为时钟源；为 5 的时候选择 ip_clk_24M 为时钟源。本章例程选择 ipg_clk 作为 GPT 定时器的时钟源，因此此位设置位 1(0b001)。

ENMODE(bit1)：GPT 使能模式，此位为 0 的时候如果关闭 GPT 定时器，计数器寄存器保存定时器关闭时候的计数值。此位为 1 的时候如果关闭 GPT 定时器，计数器寄存器就会清零。

EN(bit)：GPT 使能位，为 1 的时候使能 GPT 定时器，为 0 的时候关闭 GPT 定时器。

接下来看一下 GPT 定时器的分频寄存器 GPTx_PR，此寄存器结构如图 20.1.1.4 所示：

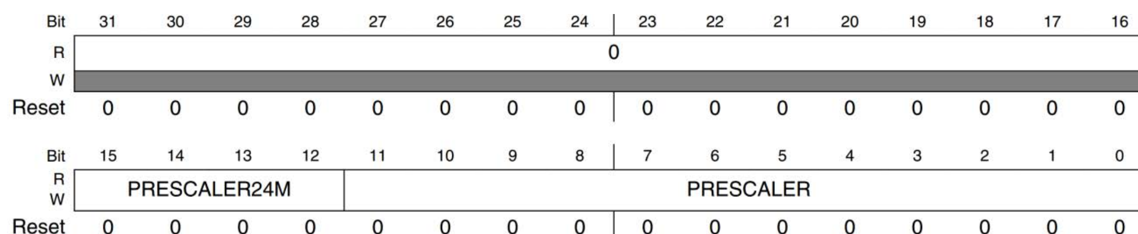


图 20.1.1.4 寄存器 GPTx_PR 寄存器

寄存器 GPTx_PR 我们用到的重要位就一个：PRESCALER(bit11:0)，这就是 12 位分频值，可设置 0~4095，分别对应 1~4096 分频。

接下来看一下 GPT 定时器的状态寄存器 GPTx_SR，此寄存器结构如图 20.1.1.5 所示：

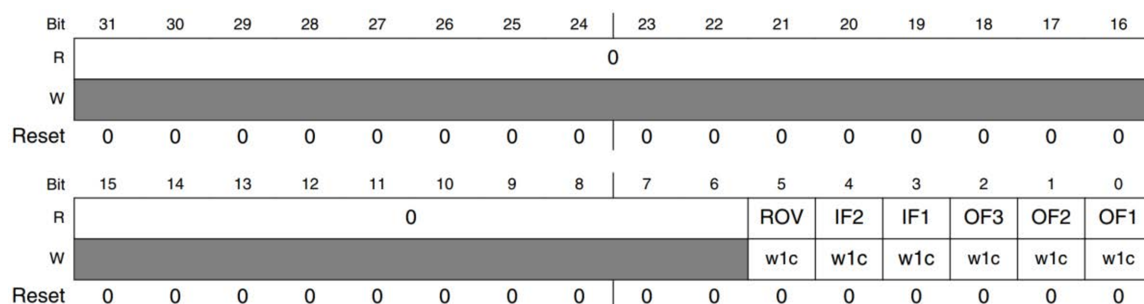


图 20.1.1.5 GPTx_SR 寄存器结构

寄存器 GPTx_SR 重要的位如下：

ROV(bit5)：回滚标志位，当计数值从 0XFFFFFFFF 回滚到 0X00000000 的时候此位置 1。

IF2~IF1(bit4:3)：输入捕获标志位，当输入捕获事件发生以后此位置 1，一共有两路输入捕获通道。如果使用输入捕获中断的话需要在中断处理函数中清除此位。

OF3~OF1(bit2:0)：输出比较中断标志位，当输出比较事件发生以后此位置 1，一共有三路输出比较通道。如果使用输出比较中断的话需要在中断处理函数中清除此位。

接着看一下 GPT 定时器的计数寄存器 GPTx_CNT，这个寄存器保存着 GPT 定时器的当前计数值。最后看一下 GPT 定时器的输出比较寄存器 GPTx_OCR，每个输出比较通道对应一个输出比较寄存器，因此一个 GPT 定时器有三个 OCR 寄存器，它们的作都是相同的。以输出比较通道 1 为例，其输出比较寄存器为 GPTx_OCR1，这是一个 32 位寄存器，用于存放 32 位的比较值。当计数器值和寄存器 GPTx_OCR1 中的值相等就会产生比较事件，如果使能了比较中断的话就会触发相应的中断。

关于 GPT 的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 1432 页的 30.6 小节。

20.1.2 定时器实现高精度延时原理

高精度延时函数的实现肯定是要借助硬件定时器，前面说了本章实验使用 GPT 定时器来实现高精度延时。如果设置 GPT 定时器的时钟源为 ipg_clk=66MHz，设置 66 分频，那么进入 GPT 定时器的最终时钟频率就是 66/66=1MHz，周期为 1us。GPT 的计数器每计一个数就表示“过去”了 1us。如果计 10 个数就表示“过去”了 10us。通过读取寄存器 GPTx_CNT 中的值就知道计了个数，比如现在要延时 100us，那么进入延时函数以后纪录下寄存器 GPTx_CNT 中的值为 200，当 GPTx_CNT 中的值为 300 的时候就表示 100us 过去了，也就是延时结束。GPTx_CNT 是个 32 位寄存器，如果时钟为 1MHz 的话，GPTx_CNT 最多可以实现 0XFFFFFFFFus=4294967295us≈4294s≈72min。也就是说 72 分钟以后 GPTx_CNT 寄存器就会回滚到 0X00000000，也就是溢出，所以需要在延时函数中要处理溢出的情况。关于定时器实现高精度延时的原理就讲解到这里，原理还是很简单的，高精度延时的实现步骤如下：

1、设置 GPT1 定时器

首先设置 GPT1_CR 寄存器的 SWR(bit15)位来复位寄存器 GPT1。复位完成以后设置寄存器 GPT1_CR 寄存器的 CLKSRC(bit8:6)位，选择 GPT1 的时钟源为 ipg_clk。设置定时器 GPT1 的工作模式，

2、设置 GPT1 的分频值

设置寄存器 GPT1_PR 寄存器的 PRESCALAR(bit111:0)位，设置分频值。

3、设置 GPT1 的比较值

如果要使用 GPT1 的输出比较中断，那么 GPT1 的输出比较寄存器 GPT1_OCR1 的值可以根据所需的中断时间来设置。本章例程不使用比较输出中断，所以将 GPT1_OCR1 设置为最大值，即：0XFFFFFFFF。

4、使能 GPT1 定时器

设置好 GPT1 定时器以后就可以使能了，设置 GPT1_CR 的 EN(bit0)位为 1 来使能 GPT1 定时器。

5、编写延时函数

GPT1 定时器已经开始运行了，可以根据前面介绍的高精度延时函数原理来编写延时函数，针对 us 和 ms 延时分别编写两个延时函数。

20.2 硬件原理分析

本试验用到的资源如下：

、一个 LED 灯：LED0。

、定时器 GPT1。

本实验通过高精度延时函数来控制 LED0 的闪烁，可以通过示波器来观察 LED0 的控制 IO 输出波形，通过波形的频率或者周期来判断延时函数精度是否正常。

20.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 12_highpreci_delay。

本章实验在上一章例程的基础上完成，更改工程名字为“delay”，直接修改 bsp_delay.c 和 bsp_delay.h 这两个文件，将 bsp_delay.h 文件改为如下所示内容：

示例代码 20.3.1 bsp_delay.h 文件代码

```
1  #ifndef __BSP_DELAY_H
2  #define __BSP_DELAY_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_delay.h
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : 延时头文件。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/1/4 左忠凯创建
12 
13              V2.0 2019/1/15 左忠凯修改
14              添加了一些函数声明。
15 ***************************************************************/
16 #include "imx6ul.h"
17 
18 /* 函数声明 */
19 void delay_init(void);
20 void delayus(unsigned int usdelay);
```

```
21 void delayms(unsigned int msdelay);
22 void delay(volatile unsigned int n);
23 void gpt1_irqhandler(void);
24
25 #endif
```

bsp_delay.h 文件就是一些函数声明，很简单。在文件 bsp_delay.c 中输入如下内容：

示例代码 20.3.2 bsp_delay.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : bsp_delay.c
作者     : 左忠凯
版本     : V1.0
描述     : 延时文件。
其他     : 无
论坛     : www.openedv.com
日志     : 初版 V1.0 2019/1/4 左忠凯创建

           V2.0 2019/1/15 左忠凯修改
           使用定时器 GPT 实现高精度延时,添加了:
           delay_init 延时初始化函数
           gpt1_irqhandler gpt1 定时器中断处理函数
           delayus us 延时函数
           delayms ms 延时函数
***************************************************************/
1   #include "bsp_delay.h"
2
3   /*
4    * @description   : 延时有关硬件初始化,主要是 GPT 定时器
5                       GPT 定时器时钟源选择 ipg_clk=66Mhz
6    * @param         : 无
7    * @return        : 无
8    */
9   void delay_init(void)
10  {
11      GPT1->CR = 0;                       /* 清零              */
12      GPT1->CR = 1 << 15;                 /* bit15 置 1 进入软复位 */
13      while((GPT1->CR >> 15) & 0x01);     /*等待复位完成         */
14
15      /*
16       * GPT 的 CR 寄存器,GPT 通用设置
17       * bit22:20 000 输出比较 1 的输出功能关闭，也就是对应的引脚没反应
18       * bit9:    0   Restart 模式,当 CNT 等于 OCR1 的时候就产生中断
19       * bit8:6   001 GPT 时钟源选择 ipg_clk=66Mhz
```

```
20       */
21      GPT1->CR = (1<<6);
22
23      /*
24       * GPT 的 PR 寄存器，GPT 的分频设置
25       * bit11:0  设置分频值，设置为 0 表示 1 分频，
26       *            以此类推，最大可以设置为 0XFFF，也就是最大 4096 分频
27       */
28      GPT1->PR = 65;  /* 66 分频，GPT1 时钟为 66M/(65+1)=1MHz */
29
30       /*
31        * GPT 的 OCR1 寄存器，GPT 的输出比较 1 比较计数值，
32        * GPT 的时钟为 1Mz，那么计数器每计一个值就是就是 1us。
33        * 为了实现较大的计数，我们将比较值设置为最大的 0XFFFFFFFF,
34        * 这样一次计满就是：0XFFFFFFFFus = 4294967296us = 4295s = 71.5min
35        * 也就是说一次计满最多 71.5 分钟，存在溢出。
36        */
37      GPT1->OCR[0] = 0XFFFFFFFF;
38      GPT1->CR |= 1<<0;          /* 使能 GPT1 */
39
40      /* 一下屏蔽的代码是 GPT 定时器中断代码，
41       * 如果想学习 GPT 定时器的话可以参考一下代码。
42       */
43  #if 0
44      /*
45       * GPT 的 PR 寄存器，GPT 的分频设置
46       * bit11:0  设置分频值，设置为 0 表示 1 分频，
47       *          以此类推，最大可以设置为 0XFFF，也就是最大 4096 分频
48       */
49
50      GPT1->PR = 65;  /* 66 分频，GPT1 时钟为 66M/(65+1)=1MHz */
51       /*
52        * GPT 的 OCR1 寄存器，GPT 的输出比较 1 比较计数值，
53        * 当 GPT 的计数值等于 OCR1 里面值时候，输出比较 1 就会发生中断
54        * 这里定时 500ms 产生中断，因此就应该为 1000000/2=500000;
55        */
56      GPT1->OCR[0] = 500000;
57
58      /*
59       * GPT 的 IR 寄存器，使能通道 1 的比较中断
60       * bit0： 0 使能输出比较中断
61       */
62      GPT1->IR |= 1 << 0;
```

```
63
64      /*
65       * 使能 GIC 里面相应的中断，并且注册中断处理函数
66       */
67      GIC_EnableIRQ(GPT1_IRQn);   /* 使能 GIC 中对应的中断 */
68      system_register_irqhandler(GPT1_IRQn,
                                   (system_irq_handler_t)gpt1_irqhandler,
                                    NULL);
69  #endif
70
71  }
72
73  #if 0
74  /* 中断处理函数 */
75  void gpt1_irqhandler(void)
76  {
77      static unsigned char state = 0;
78      state = !state;
79      /*
80       * GPT 的 SR 寄存器，状态寄存器
81       * bit2： 1 输出比较 1 发生中断
82       */
83      if(GPT1->SR & (1<<0))
84      {
85          led_switch(LED2, state);
86      }
87      GPT1->SR |= 1<<0; /* 清除中断标志位 */
88  }
89  #endif
90
91  /*
92   * @description    : 微秒(us)级延时
93   * @param - value : 需要延时的 us 数,最大延时 0XFFFFFFFFus
94   * @return         : 无
95   */
96  void delayus(unsigned    int usdelay)
97  {
98      unsigned long oldcnt,newcnt;
99      unsigned long tcntvalue = 0;    /* 走过的总时间  */
100
101     oldcnt = GPT1->CNT;
102     while(1)
103     {
```

```
104         newcnt = GPT1->CNT;
105         if(newcnt != oldcnt)
106         {
107             if(newcnt > oldcnt)      /* GPT 是向上计数器,并且没有溢出 */
108                 tcntvalue += newcnt - oldcnt;
109             else                     /* 发生溢出                      */
110                 tcntvalue += 0XFFFFFFFF-oldcnt + newcnt;
111             oldcnt = newcnt;
112             if(tcntvalue >= usdelay) /* 延时时间到了                  */
113             break;                   /*  跳出                         */
114         }
115     }
116 }
117
118 /*
119  * @description      : 毫秒(ms)级延时
120  * @param - msdelay  : 需要延时的 ms 数
121  * @return           : 无
122  */
123 void delayms(unsigned int msdelay)
124 {
125     int i = 0;
126     for(i=0; i<msdelay; i++)
127     {
128         delayus(1000);
129     }
130 }
131
132 /*
133  * @description   : 短时间延时函数
134  * @param - n     : 要延时循环次数(空操作循环次数，模式延时)
135  * @return        : 无
136  */
137 void delay_short(volatile unsigned int n)
138 {
139     while(n--){}
140 }
141
142 /*
143  * @description   : 延时函数,在 396Mhz 的主频下
144  *                 延时时间大约为 1ms
145  * @param - n     : 要延时的 ms 数
146  * @return        : 无
```

```
147  */
148 void delay(volatile unsigned int n)
149 {
150     while(n--)
151     {
152         delay_short(0x7ff);
153     }
154 }
```

文件 bsp_delay.c 中一共有 5 个函数，分别为：delay_init、delayus、delayms 、delay_short 和 delay。除了 delay_short 和 delay 以外，其他三个都是新增加的。函数 delay_init 是延时初始化函数，主要用于初始化 GPT1 定时器，设置其时钟源、分频值和输出比较寄存器值。第 43 到 68 行被屏蔽掉的程序是 GPT1 的中断初始化代码，如果要使用 GPT1 的中断功能的话可以参考此部分代码。第 73 到 89 行被屏蔽掉的程序是 GPT1 的中断处理函数 gpt1_irqhandler，同样的，如果需要使用 GPT1 中断功能的话可以参考此部分代码。

函数 delayus 和 delayms 就是 us 级和 ms 级的高精度延时函数，函数 delayus 就是按照我们在 20.1.2 小节讲解的高精度延时原理编写的，delayus 函数处理 GPT1 计数器溢出的情况。函数 delayus 只有一个参数 usdelay，这个参数就是要延时的 us 数。delayms 函数很简单，就是对 delayus(1000)的多次叠加，此函数也只有一个参数 msdelay，也就是要延时的 ms 数。

最后修改 mian.c 文件，内容如下：

示例代码 20.3.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    :    mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 12 高精度延时实验
其他      : 本实验我们学习如何使用 I.MX6U 的 GPT 定时器来实现高精度延时，
            以前的延时都是靠空循环来实现的，精度很差，只能用于要求
            不高的场合。使用 I.MX6U 的硬件定时器就可以实现高精度的延时，
            最低可以做到 20us 的高精度延时。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_keyfilter.h"
8
9  /*
10  * @description  : main 函数
```

```
11  * @param        : 无
12  * @return       : 无
13  */
14 int main(void)
15 {
16     unsigned char state = OFF;
17 
18     int_init();              /* 初始化中断(一定要最先调用！) */
19     imx6u_clkinit();         /* 初始化系统时钟              */
20     delay_init();            /* 初始化延时                  */
21     clk_enable();            /* 使能所有的时钟              */
22     led_init();              /* 初始化 led                  */
23     beep_init();             /* 初始化 beep                 */
24 
25     while(1)
26     {
27         state = !state;
28         led_switch(LED0, state);
29         delayms(500);
30     }
31 
32     return 0;
33 }
```

main.c 函数很简单，在第 20 行调用 delay_init 函数进行延时初始化，最后在 while 循环中周期性的点亮和熄灭 LED0，调用函数 delayms 来实现延时。

20.4 编译下载验证

20.4.1 编写 Makefile 和链接脚本

因为本章例程并没有新建任何文件，所以只需要修改 Makefile 中的 TARGET 为 delay 即可，链接脚本报纸不变。

20.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 delay.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload delay.bin /dev/sdd     //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。程序运行正常的话 LED0 会以 500ms 为周期不断的亮、灭闪烁。可以通过肉眼观察 LED 亮灭的时间是否为 500ms。但是肉眼观察肯定不准确，既然本章号称高精度延时实验，那么就得经得住专业仪器的测试。我们将“示例代码 20.3.3”中第 29 行，也就是 mian 函数 while 循环中的延时改为“delayus(20)”，也就是 LED0 亮灭的时间各为 20us，那么一个完整的周期就是 20+20=40us，LED0 对应的 IO 频

率就应该是 1/0.00004=25000Hz=25KHz。使用示波器测试 LED0 对应的 IO 频率，结果如图 20.4.3.1 所示：

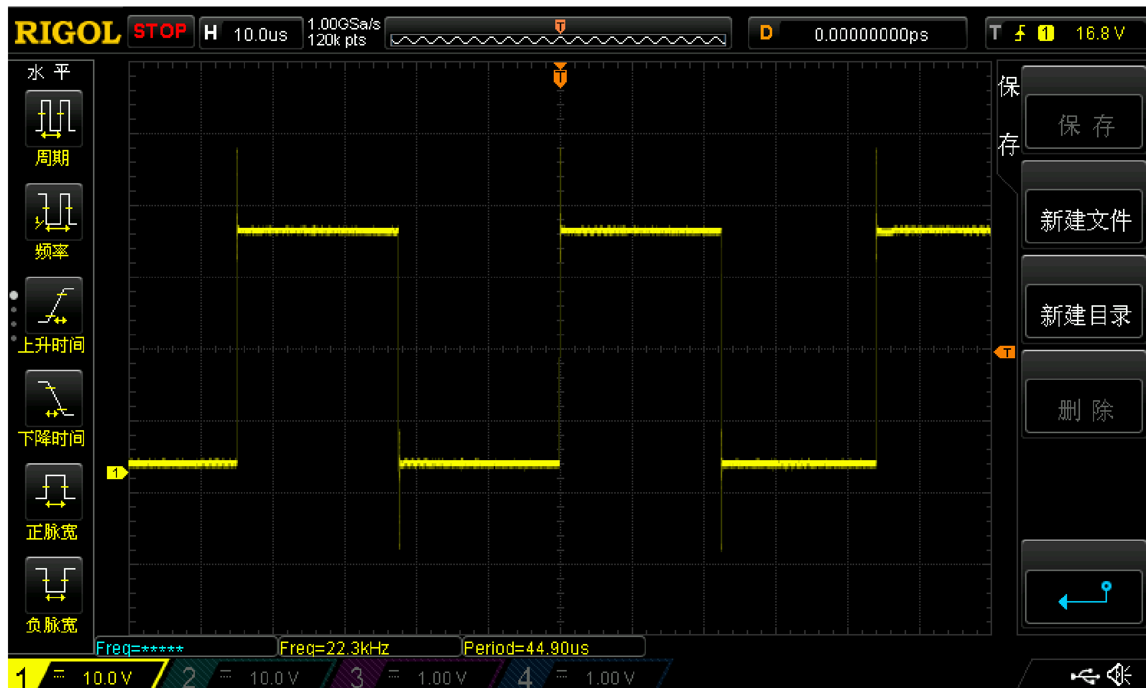


图 20.4.3.1 20us 延时波形

从图 20.4.3.1 可以看出，LED0 对应的 IO 波形频率为 22.3KHz，周期是 44.9us，那么 main 函数中 while 循环执行一次的时间就是 44.9/2=22.45us，大于我们设置的 20us，看起来好像是延时不准确。但是我们要知道这 22.45us 是 main 函数里面 while 循环总执行时间，也就是下面代码的总执行时间：

```
while(1)
{
    state = !state;
    led_switch(LED0, state);
    delayus(20);
}
```

在上面代码中不止有 delayus(20)延时函数，还有控制 LED 灯亮灭的函数，这些代码的执行也需要时间的，即使是 delayus 函数，其内部也是要消耗一些时间的。假如我们将 while 循环里面的代码改为如下形式：

```
while(1)
{
    GPIO1->DR &= ~(1<<3);
    delayus(20);
    GPIO1->DR |= (1<<3);
    delayus(20);
}
```

上述代码我们通过直接操作寄存器的方式来控制 IO 输出高低电平，理论上 while 循环执行时间会更小，并且 while 循环里面使用了两个 delayus(20)，因此执行一次 while 循环的理论时间应该是 40us，和上面做的实验一样。重新使用示波器测量一下，结果如图 20.4.3.2 所示：

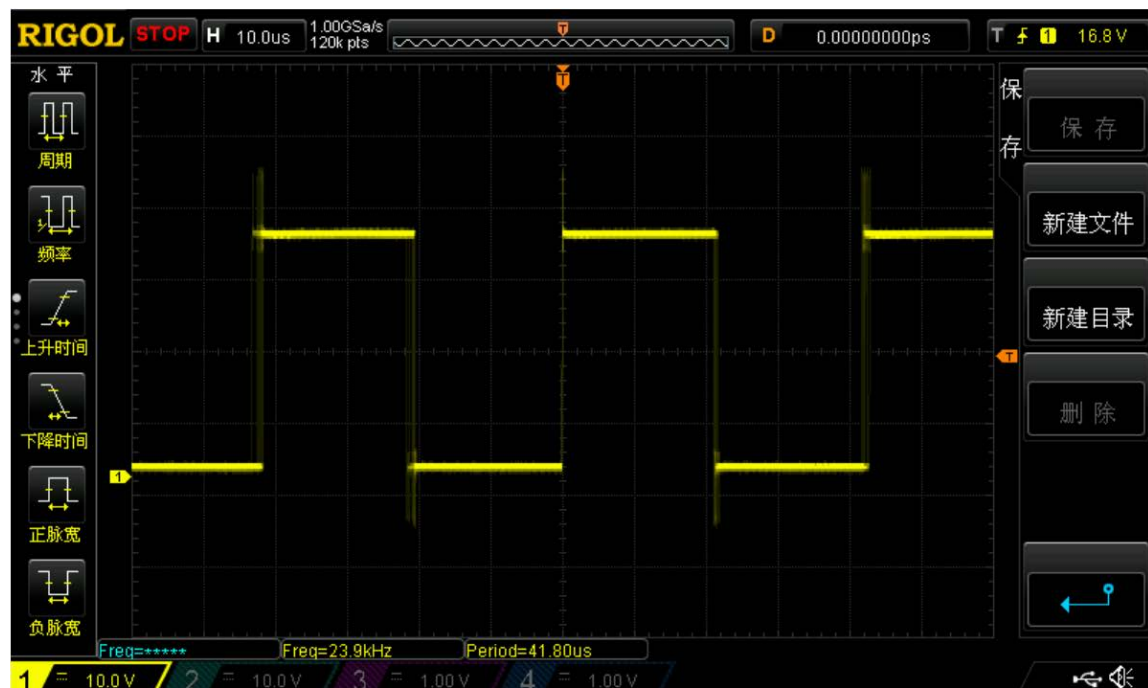


图 20.4.3.2 修改 while 循环后的波形

从图 20.4.3.2 可以看出，此时 while 循环执行一次的时间是 41.8us，那么一次 delayus(20)的时间就是 41.8/2=20.9us，很接近我们的 20us 理论值。但是还是因为有其他程序开销存在，在加上示波器测量误差，所以不可能测量出绝对的 20us。但是其已经非常接近了，基本可以证明我们的高精度延时函数是成功的、可以用的。

第二十一章　UART 串口通信实验

不管是单片机开发还是嵌入式 Linux 开发，串口都是最常用到的外设。可以通过串口将开发板与电脑相连，然后在电脑上通过串口调试助手来调试程序。还有很多的模块，比如蓝牙、GPS、GPRS 等都使用的串口来与主控进行通信的，在嵌入式 Linux 中一般使用串口作为控制台，所以掌握串口是必备的技能。本章我们就来学习如何驱动 I.MX6U 上的串口，并使用串口和电脑进行通信。

21.1 I.MX6U 串口简介

21.1.1 UART 简介

1、UART 通信格式

串口全称叫做串行接口，通常也叫做 COM 接口，串行接口指的是数据一个一个的顺序传输，通信线路简单。使用两条线即可实现双向通信，一条用于发送，一条用于接收。串口通信距离远，但是速度相对会低，串口是一种很常用的工业接口。I.MX6U 自带的 UART 外设就是串口的一种，UART 全称是 Universal Asynchronous Receiver/Trasmitter，也就是异步串行收发器。既然有异步串行收发器，那肯定也有同步串行收发器，学过 STM32 的同学应该知道，STM32 除了有 UART 外，还有另外一个叫做 USART 的东西。USART 的全称是 Universal Synchronous/Asynchronous Receiver/Transmitter，也就是同步/异步串行收发器。相比 UART 多了一个同步的功能，在硬件上体现出来的就是多了一条时钟线。一般 USART 是可以作为 UART 使用的，也就是不使用其同步的功能。

UART 作为串口的一种，其工作原理也是将数据一位一位的进行传输，发送和接收个用一条线，因此通过 UART 接口与外界相连最少只需要三条线：TXD(发送)、RXD(接收)和 GND(地线)。图 21.1.1.1 就是 UART 的通信格式：

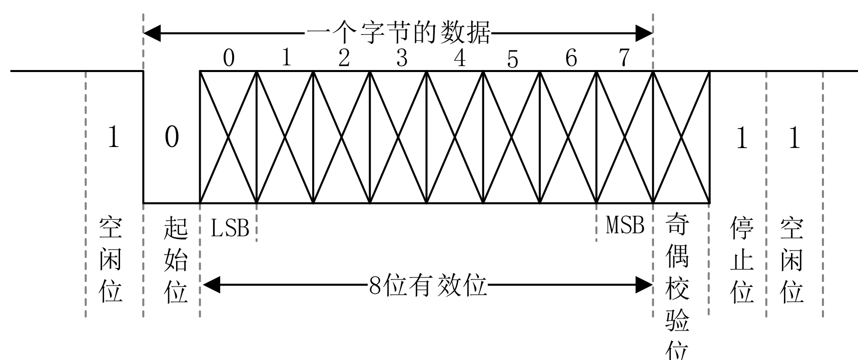


图 21.1.1.1 UART 通信格式

图 21.1.1.1 中各位的含义如下：

空闲位：数据线在空闲状态的时候为逻辑“1”状态，也就是高电平，表示没有数据线空闲，没有数据传输。

起始位：当要传输数据的时候先传输一个逻辑“0”，也就是将数据线拉低，表示开始数据传输。

数据位：数据位就是实际要传输的数据，数据位数可选择 5~8 位，我们一般都是按照字节传输数据的，一个字节 8 位，因此数据位通常是 8 位的。低位在前，先传输，高位最后传输。

奇偶校验位：这是对数据中“1”的位数进行奇偶校验用的，可以不使用奇偶校验功能。

停止位：数据传输完成标志位，停止位的位数可以选择 1 位、1.5 位或 2 位高电平，一般都选择 1 位停止位。

波特率：波特率就是 UART 数据传输的速率，也就是每秒传输的数据位数，一般选择 9600、19200、115200 等。

2、UART 电平标准

UART 一般的接口电平有 TTL 和 RS-232，一般开发板上都有 TXD 和 RXD 这样的引脚，这些引脚低电平表示逻辑 0，高电平表示逻辑 1，这个就是 TTL 电平。RS-232 采用差分线，-3~-15V 表示逻辑 1，+3~+15V 表示逻辑 0。一般图 21.1.1.2 中的接口就是 TTL 电平：



图 21.1.1.2 TTL 电平接口

图 21.1.1.2 中的模块就是 USB 转 TTL 模块，TTL 接口部分有 VCC、GND、RXD、TXD、RTS 和 CTS。RTS 和 CTS 基本用不到，使用的时候通过杜邦线和其他模块的 TTL 接口相连即可。

RS-232 电平需要 DB9 接口，I.MX6U-ALPHA 开发板上的 COM3(UART3)口就是 RS-232 接口的，如图 21.1.1.3 所示：



图 21.1.1.3 I.MX6U-ALPHA 开发板 RS-232 接口

由于现在的电脑都没有 DB9 接口了，取而代之的是 USB 接口，所以就催生出了很多 USB 转串口 TTL 芯片，比如 CH340、PL2303 等。通过这些芯片就可以实现串口 TTL 转 USB。I.MX6U-ALPHA 开发板就使用 CH340 芯片来完成 UART1 和电脑之间的连接，只需要一条 USB 线即可，如图 21.1.1.4 所示。

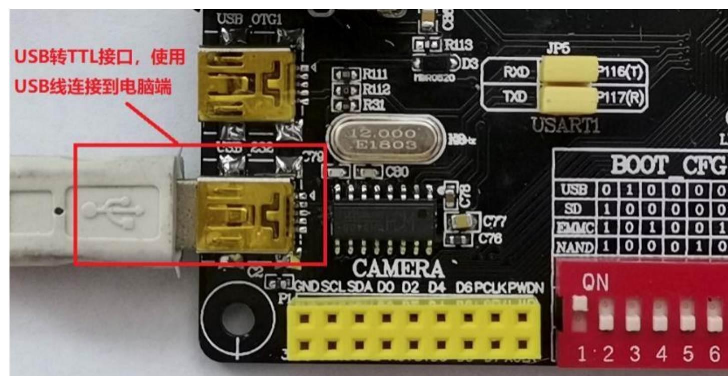


图 21.1.1.4 I.MX6U-ALPHA 开发板 USB 转 TTL 接口

21.1.2 I.MX6U UART 简介

上一小节介绍了 UART 接口，本小节来具体看一下 I.MX6U 的 UART 接口，I.MX6U 一共有 8 个 UART，其主要特性如下：

、兼容 TIA/EIA-232F 标准，速度最高可到 5Mbit/S。
、支持串行 IR 接口，兼容 IrDA，最高可到 115.2Kbit/s。
、支持 9 位或者多节点模式(RS-485)。
、1 或 2 位停止位。
、可编程的奇偶校验(奇校验和偶校验)。
、自动波特率检测(最高支持 115.2Kbit/S)。

I.MX6U 的 UART 功能很多，但是我们本章就只用到其最基本的串口功能，关于 UART 其它功能的介绍请参考《I.MX6ULL 参考手册》第 3561 页的“Chapter 55 Universal Asynchronous Receiver/Transmitter(UART)”章节。

UART 的时钟源是由寄存器 CCM_CSCDR1 的 UART_CLK_SEL(bit)位来选择的，当为 0 的时候 UART 的时钟源为 pll3_80m(80MHz)，如果为 1 的时候 UART 的时钟源为 osc_clk(24M)，一般选择 pll3_80m 作为 UART 的时钟源。寄存器 CCM_CSCDR1 的 UART_CLK_PODF(bit5:0)位是 UART 的时钟分频值，可设置 0~63，分别对应 1~64 分频，一般设置为 1 分频，因此最终进入 UART 的时钟为 80MHz。

接下来看一下 UART 几个重要的寄存器，第一个就是 UART 的控制寄存器 1，即 UARTx_UCR1(x=1~8)，此寄存器的结构如图 21.1.2.1 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	ADEN	ADBR	TRDYEN	IDEN	ICD		RRDYEN	RXDMAEN	IREN	TXMPTYEN	RTSDEN	SNDBRK	TXDMAEN	ATDMAEN	DOZE	UARTEN
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 21.1.2.1 寄存器 UARTx_UCR1 结构

寄存器 UARTx_UCR1 我们用到的重要位如下：

ADBR(bit14)：自动波特率检测使能位，为 0 的时候关闭自动波特率检测，为 1 的时候使能自动波特率检测。

UARTEN(bit0)：UART 使能位，为 0 的时候关闭 UART，为 1 的时候使能 UART。

接下来看一下 UART 的控制寄存器 2，即：UARTx_UCR2，此寄存器结构如图 21.1.2.2 所示：

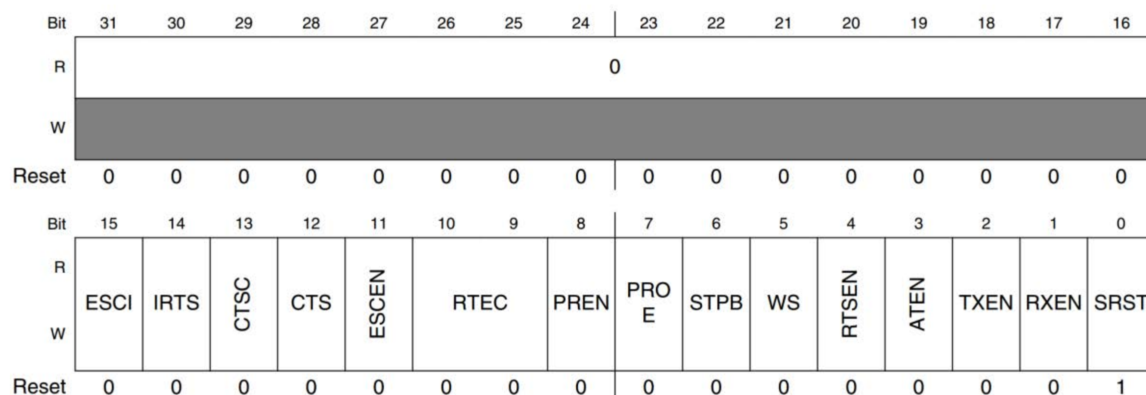


图 21.1.2.2 寄存器 UARTx_UCR2 结构

寄存器 UARTx_UCR2 用到的重要位如下：

IRTS(bit14)：为 0 的时候使用 RTS 引脚功能，为 1 的时候忽略 RTS 引脚。

PREN(bit8)：奇偶校验使能位，为 0 的时候关闭奇偶校验，为 1 的时候使能奇偶校验。

PROE(bit7)：奇偶校验模式选择位，开启奇偶校验以后此位如果为 0 的话就使用偶校验，此位为 1 的话就使能奇校验。

STOP(bit6)：停止位数量，为 0 的话 1 位停止位，为 1 的话 2 位停止位。

WS(bit5)：数据位长度，为 0 的时候选择 7 位数据位，为 1 的时候选择 8 位数据位。

TXEN(bit2)：发送使能位，为 0 的时候关闭 UART 的发送功能，为 1 的时候打开 UART 的发送功能。

RXEN(bit1)：接收使能位，为 0 的时候关闭 UART 的接收功能，为 1 的时候打开 UART 的接收功能。

SRST(bit0)：软件复位，为 0 的是时候软件复位 UART，为 1 的时候表示复位完成。复位完成以后此位会自动置 1，表示复位完成。此位只能写 0，写 1 会被忽略掉。

接下来看一下 UARTx_UCR3 寄存器，此寄存器结构如图 21.1.2.3 所示：

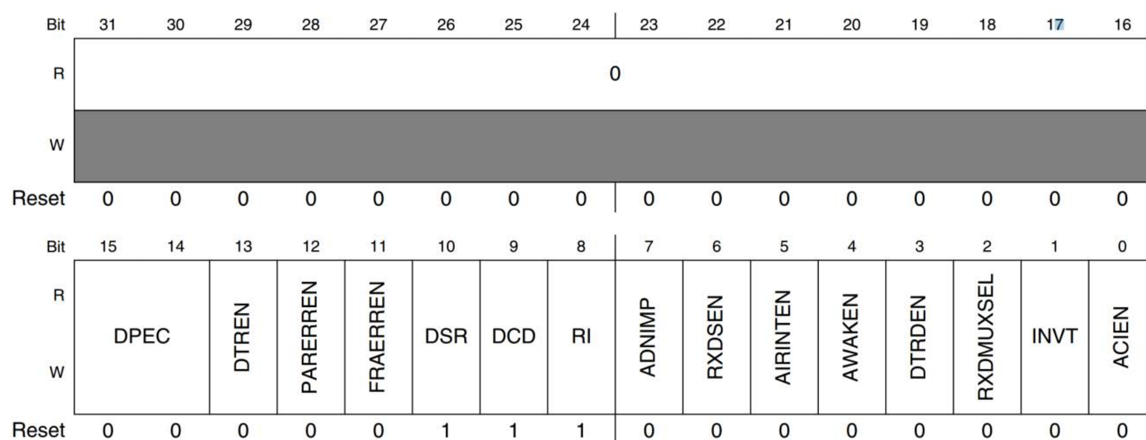


图 21.1.2.3 UARTx_UCR3 寄存器结构体

本章实验就用到了寄存器 UARTx_UCR3 中的位 RXDMUXSEL(bit2)，这个位应该始终为 1，找个在《I.MX6ULL 参考手册》第 3624 页有说明。

接下来看一下寄存器 UARTx_USR2，这个是 UART 的状态寄存器 2，此寄存器结构如图 21.1.2.4 所示：

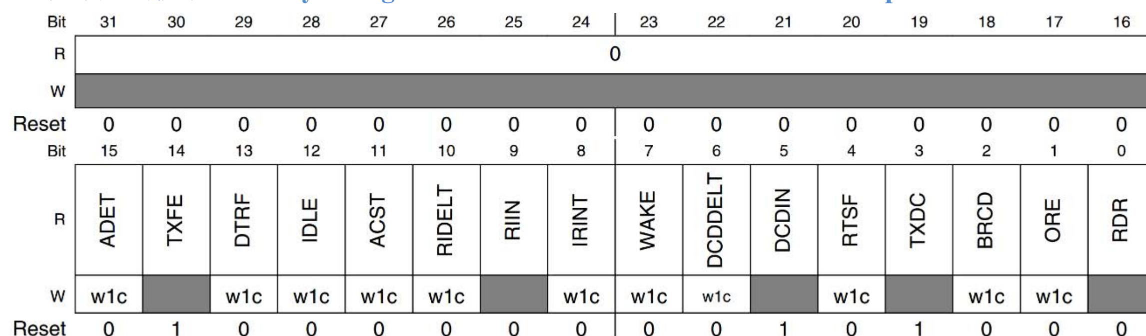


图 21.1.2.4 寄存器 UARTx_USR2 结构

寄存器 UARTx_USR2 用到的重要位如下：

TXDC(bit3)：发送完成标志位，为 1 的时候表明发送缓冲(TxFIFO)和移位寄存器为空，也就是发送完成，向 TxFIFO 写入数据此位就会自动清零。

RDR(bit0)：数据接收标志位，为 1 的时候表明至少接收到一个数据，从寄存器 UARTx_URXD 读取数据接收到的数据以后此为会自动清零。

接下来看一下寄存器 UARTx_UFCR、UARTx_UBIR 和 UARTx_UBMR，寄存器 UARTx_UFCR 中我们要用到的是位 RFDIV(bit9:7)，用来设置参考时钟分频，设置如表 21.1.2.1 所示：

RFDIV(bit9:7)	分频值
000	6 分频
001	5 分频
010	4 分频
011	3 分频
100	2 分频
101	1 分频
110	7 分频
111	保留

表 21.1.2.1 RFDIV 分频表

通过这三个寄存器可以设置 UART 的波特率，波特率的计算公式如下：

$$\text{Baud Rate} = \frac{Ref\ Freq}{(16 \times \frac{UBMR\ +\ 1}{UBIR\ +\ 1})}$$

Ref Freq：经过分频以后进入 UART 的最终时钟频率。

UBMR：寄存器 UARTx_UBMR 中的值。

UBIR：寄存器 UARTx_UBIR 中的值。

通过 UARTx_UFCR 的 RFDIV 位、UARTx_UBMR 和 UARTx_UBIR 这三者的配合即可得到我们想要的波特率。比如现在要设置 UART 波特率为 115200，那么可以设置 RFDIV 为 5(0b101)，也就是 1 分频，因此 Ref Freq=80MHz。设置 UBIR=71，UBMR=3124，根据上面的公式可以得到：

$$\text{Baud Rate} = \frac{Ref\ Freq}{(16 \times \frac{UBMR+1}{UBIR+1})} = \frac{80000000}{(16 \times \frac{3124+1}{71+1})} = 115200$$

最后来看一下寄存器 UARTx_URXD 和 UARTx_UTXD，这两个寄存器分别为 UART 的接收和发送数据寄存器，这两个寄存器的低八位为接收到的和要发送的数据。读取寄存器

UARTx_URXD 即可获取到接收到的数据，如果要通过 UART 发送数据，直接将数据写入到寄存器 UARTx_UTXD 即可。

关于 UART 的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 3608 页的 55.15 小节。本章我们使用 I.MX6U 的 UART1 来完成开发板与电脑串口调试助手之间串口通信， UART1 的配置步骤如下：

1、设置 UART1 的时钟源

设置 UART 的时钟源为 pll3_80m，设置寄存器 CCM_CSCDR1 的 UART_CLK_SEL 位为 0 即可。

2、初始化 UART1

初始化 UART1 所使用 IO，设置 UART1 的寄存器 UART1_UCR1~UART1_UCR3，设置内容包括波特率，奇偶校验、停止位、数据位等等。

4、使能 UART1

UART1 初始化完成以后就可以使能 UART1 了，设置寄存器 UART1_UCR1 的位 UARTEN 为 1。

5、编写 UART1 数据收发函数

编写两个函数用于 UART1 的数据收发操作。

21.2 硬件原理分析

本试验用到的资源如下：

、一个 LED 灯：LED0。

、串口 1。

I.MX6U-ALPHA 开发板串口 1 硬件原理图如图 22.2.1 所示：

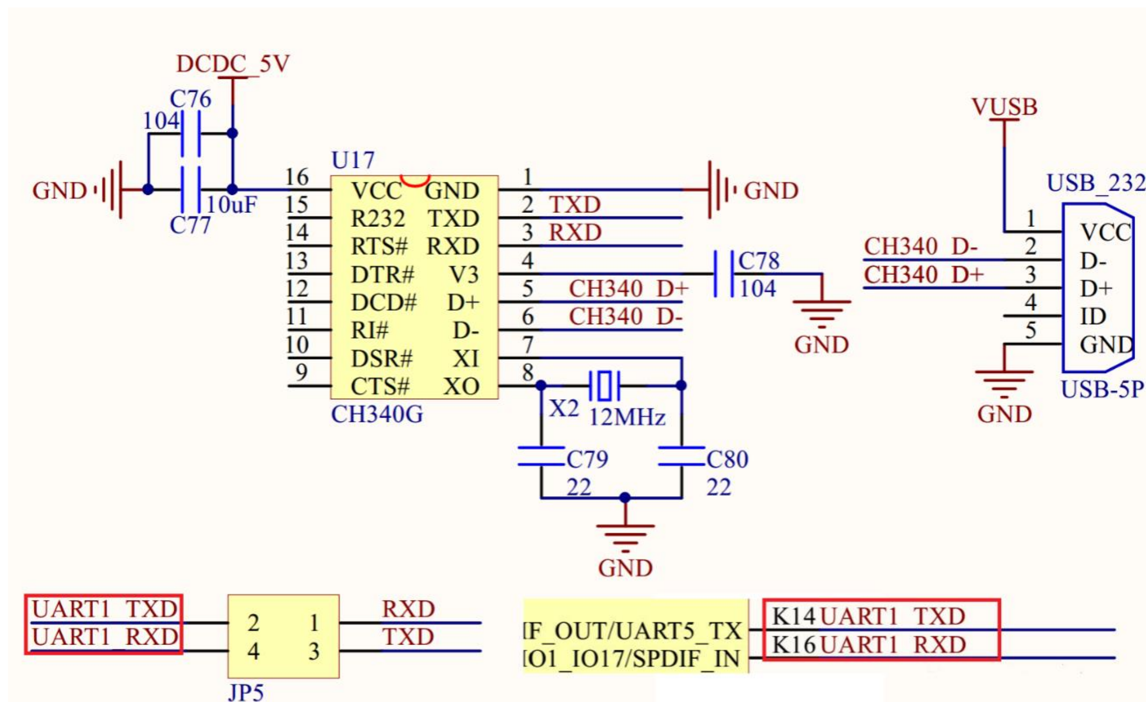


图 22.2.1 I.MX6U-ALPHA 开发板串口 1 原理图

在做实验之前需要用 USB 串口线将串口 1 和电脑连接起来，并且还需要设置 JP5 跳线帽，

将串口 1 的 RXD、TXD 两个引脚分别于 P116、P117 连接一起，如图 22.2.2 所示：

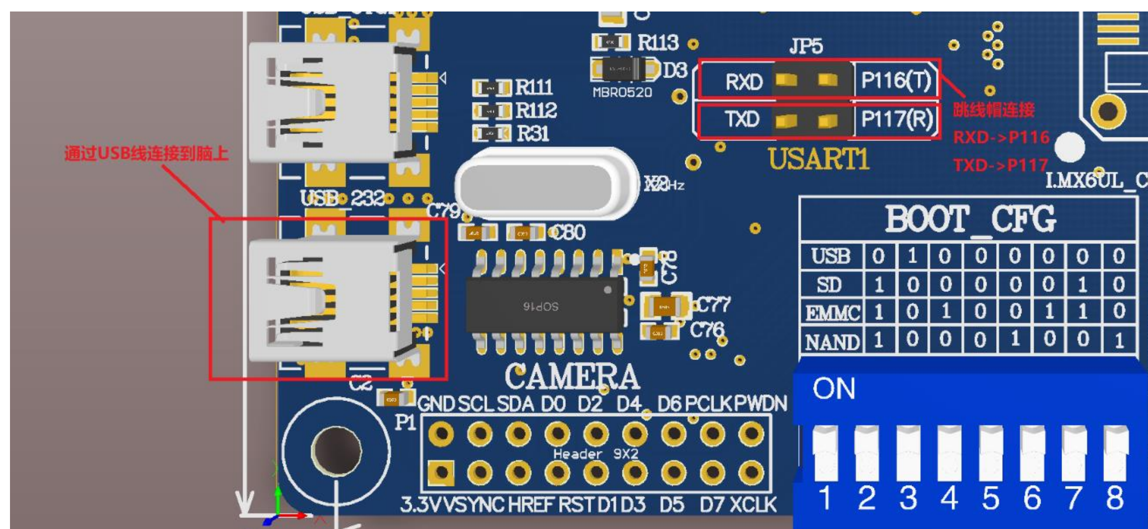


图 22.2.2 串口 1 硬件连接设置图

硬件连接设置好以后就可以开始软件编写了，本章实验我们初始化好 UART1，然后等待 SecureCRT 给开发板发送一个字节的数据，开发板接收到 SecureCRT 发送过来的数据以后在同通过串口 1 发送给 SecureCRT。

21.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 13_uart。

本章实验在上一章例程的基础上完成，更改工程名字为“uart”，然后在 bsp 文件夹下创建名为“uart”的文件夹，然后在 bsp/uart 中新建 bsp_uart.c 和 bsp_uart.h 这两个文件。在 bsp_uart.h 中输入如下内容：

示例代码 21.3.1 bsp_uart.h 文件代码

```
1  #ifndef _BSP_UART_H
2  #define _BSP_UART_H
3  #include "imx6ul.h"
4  /***************************************************************
5  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
6  文件名     : bsp_uart.h
7  作者       : 左忠凯
8  版本       : V1.0
9  描述       : 串口驱动文件头文件。
10 其他       : 无
11 论坛       : www.openedv.com
12 日志       : 初版 V1.0 2019/1/15 左忠凯创建
13 ***************************************************************/
14 
15 /* 函数声明 */
16 void uart_init(void);
17 void uart_io_init(void);
18 void uart_disable(UART_Type *base);
```

```
19 void uart_enable(UART_Type *base);
20 void uart_softreset(UART_Type *base);
21 void uart_setbaudrate(UART_Type *base,
                              unsigned int baudrate,
                              unsigned int srcclock_hz);
22 void putc(unsigned char c);
23 void puts(char *str);
24 unsigned char getc(void);
25 void raise(int sig_nr);
26
27 #endif
```

文件 bsp_uart.h 内容很简单，就是一些函数声明。继续在文件 bsp_uart.c 中输入如下所示内容：

示例代码 21.3.2 bsp_uart.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_uart.c
作者      : 左忠凯
版本      : V1.0
描述      : 串口驱动文件。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1   #include "bsp_uart.h"
2
3   /*
4    * @description   : 初始化串口 1,波特率为 115200
5    * @param         : 无
6    * @return        : 无
7    */
8   void uart_init(void)
9   {
10      /* 1、初始化串口 IO */
11      uart_io_init();
12
13      /* 2、初始化 UART1  */
14      uart_disable(UART1);         /* 先关闭 UART1         */
15      uart_softreset(UART1);       /* 软件复位 UART1       */
16
17      UART1->UCR1 = 0;             /* 先清除 UCR1 寄存器    */
18      UART1->UCR1 &= ~(1<<14);     /* 关闭自动波特率检测    */
19
```

```
20      /*
21       * 设置 UART 的 UCR2 寄存器，设置字长，停止位，校验模式，关闭硬件流控
22       * bit14: 1 忽略 RTS 引脚
23       * bit8:  0 关闭奇偶校验
24       * bit6:  0 1 位停止位
25       * bit5:  1 8 位数据位
26       * bit2:  1 打开发送
27       * bit1:  1 打开接收
28       */
29      UART1->UCR2 |= (1<<14) | (1<<5) | (1<<2) | (1<<1);
30      UART1->UCR3 |= 1<<2;        /* UCR3 的 bit2 必须为 1 */
31  
32      /*
33       * 设置波特率
34       * 波特率计算公式:Baud Rate = Ref Freq / (16 * (UBMR + 1)/(UBIR+1))
35       * 如果要设置波特率为 115200，那么可以使用如下参数:
36       * Ref Freq = 80M 也就是寄存器 UFCR 的 bit9:7=101, 表示 1 分频
37       * UBMR = 3124
38       * UBIR =  71
39       * 因此波特率= 80000000/(16 * (3124+1)/(71+1))
40       *           = 80000000/(16 * 3125/72)
41       *           = (80000000*72) / (16*3125)
42       *           = 115200
43       */
44      UART1->UFCR = 5<<7;         /* ref freq 等于 ipg_clk/1=80Mhz */
45      UART1->UBIR = 71;
46      UART1->UBMR = 3124;
47  
48  #if 0
49      uart_setbaudrate(UART1, 115200, 80000000); /* 设置波特率 */
50  #endif
51  
52      uart_enable(UART1); /* 使能串口 */
53  }
54  
55  /*
56   * @description    : 初始化串口 1 所使用的 IO 引脚
57   * @param          : 无
58   * @return         : 无
59   */
60  void uart_io_init(void)
61  {
62      /* 1、初始化串口 IO
```

```
63       * UART1_RXD -> UART1_TX_DATA
64       * UART1_TXD -> UART1_RX_DATA
65       */
66      IOMUXC_SetPinMux(IOMUXC_UART1_TX_DATA_UART1_TX, 0);
67      IOMUXC_SetPinMux(IOMUXC_UART1_RX_DATA_UART1_RX, 0);
68      IOMUXC_SetPinConfig(IOMUXC_UART1_TX_DATA_UART1_TX, 0x10B0);
69      IOMUXC_SetPinConfig(IOMUXC_UART1_RX_DATA_UART1_RX, 0x10B0);
70  }
71
72  /*
73   * @description          : 波特率计算公式,
74   *                          可以用此函数计算出指定串口对应的 UFCR,
75   *                          UBIR 和 UBMR 这三个寄存器的值
76   * @param - base          : 要计算的串口。
77   * @param - baudrate      : 要使用的波特率。
78   * @param - srcclock_hz   : 串口时钟源频率，单位 Hz
79   * @return                : 无
80   */
81  void uart_setbaudrate(UART_Type *base,
                          unsigned int baudrate,
                          unsigned int srcclock_hz)
82  {
83      uint32_t numerator = 0u;
84      uint32_t denominator = 0U;
85      uint32_t divisor = 0U;
86      uint32_t refFreqDiv = 0U;
87      uint32_t divider = 1U;
88      uint64_t baudDiff = 0U;
89      uint64_t tempNumerator = 0U;
90      uint32_t tempDenominator = 0u;
91
92      /* get the approximately maximum divisor */
93      numerator = srcclock_hz;
94      denominator = baudrate << 4;
95      divisor = 1;
96
97      while (denominator != 0)
98      {
99          divisor = denominator;
100         denominator = numerator % denominator;
101         numerator = divisor;
102     }
103
```

```
104     numerator = srcclock_hz / divisor;
105     denominator = (baudrate << 4) / divisor;
106 
107     /* numerator ranges from 1 ~ 7 * 64k */
108     /* denominator ranges from 1 ~ 64k */
109     if ((numerator > (UART_UBIR_INC_MASK * 7)) || (denominator >
                                                   UART_UBIR_INC_MASK))
110     {
111         uint32_t m = (numerator - 1) / (UART_UBIR_INC_MASK * 7) + 1;
112         uint32_t n = (denominator - 1) / UART_UBIR_INC_MASK + 1;
113         uint32_t max = m > n ? m : n;
114         numerator /= max;
115         denominator /= max;
116         if (0 == numerator)
117         {
118             numerator = 1;
119         }
120         if (0 == denominator)
121         {
122             denominator = 1;
123         }
124     }
125     divider = (numerator - 1) / UART_UBIR_INC_MASK + 1;
126 
127     switch (divider)
128     {
129         case 1:
130             refFreqDiv = 0x05;
131             break;
132         case 2:
133             refFreqDiv = 0x04;
134             break;
135         case 3:
136             refFreqDiv = 0x03;
137             break;
138         case 4:
139             refFreqDiv = 0x02;
140             break;
141         case 5:
142             refFreqDiv = 0x01;
143             break;
144         case 6:
145             refFreqDiv = 0x00;
```

```
146             break;
147         case 7:
148             refFreqDiv = 0x06;
149             break;
150         default:
151             refFreqDiv = 0x05;
152             break;
153     }
154     /* Compare the difference between baudRate_Bps and calculated
155      * baud rate. Baud Rate = Ref Freq / (16 * (UBMR + 1)/(UBIR+1)).
156      * baudDiff = (srcClock_Hz/divider)/( 16 * ((numerator /
                                                divider)/ denominator).
157      */
158     tempNumerator = srcclock_hz;
159     tempDenominator = (numerator << 4);
160     divisor = 1;
161     /* get the approximately maximum divisor */
162     while (tempDenominator != 0)
163     {
164         divisor = tempDenominator;
165         tempDenominator = tempNumerator % tempDenominator;
166         tempNumerator = divisor;
167     }
168     tempNumerator = srcclock_hz / divisor;
169     tempDenominator = (numerator << 4) / divisor;
170     baudDiff = (tempNumerator * denominator) / tempDenominator;
171     baudDiff = (baudDiff >= baudrate) ? (baudDiff - baudrate) :
                                                 (baudrate - baudDiff);
172 
173     if (baudDiff < (baudrate / 100) * 3)
174     {
175         base->UFCR &= ~UART_UFCR_RFDIV_MASK;
176         base->UFCR |= UART_UFCR_RFDIV(refFreqDiv);
177         base->UBIR = UART_UBIR_INC(denominator - 1);
178         base->UBMR = UART_UBMR_MOD(numerator / divider - 1);
179     }
180 }
181 
182 /*
183  * @description   : 关闭指定的 UART
184  * @param - base  : 要关闭的 UART
185  * @return        : 无
186  */
```

```
187 void uart_disable(UART_Type *base)
188 {
189     base->UCR1 &= ~(1<<0);
190 }
191
192 /*
193  * @description  : 打开指定的 UART
194  * @param - base : 要打开的 UART
195  * @return       : 无
196  */
197 void uart_enable(UART_Type *base)
198 {
199     base->UCR1 |= (1<<0);
200 }
201
202 /*
203  * @description  : 复位指定的 UART
204  * @param - base : 要复位的 UART
205  * @return       : 无
206  */
207 void uart_softreset(UART_Type *base)
208 {
209     base->UCR2 &= ~(1<<0);              /* 复位 UART   */
210     while((base->UCR2 & 0x1) == 0);     /* 等待复位完成 */
211 }
212
213 /*
214  * @description  : 发送一个字符
215  * @param - c    : 要发送的字符
216  * @return       : 无
217  */
218 void putc(unsigned char c)
219 {
220     while(((UART1->USR2 >> 3) &0X01) == 0);/* 等待上一次发送完成  */
221     UART1->UTXD = c & 0XFF;                /* 发送数据          */
222 }
223
224 /*
225  * @description  : 发送一个字符串
226  * @param - str  : 要发送的字符串
227  * @return       : 无
228  */
229 void puts(char *str)
```

```
230 {
231     char *p = str;
232
233     while(*p)
234         putc(*p++);
235 }
236
237 /*
238  * @description   : 接收一个字符
239  * @param         : 无
240  * @return        : 接收到的字符
241  */
242 unsigned char getc(void)
243 {
244     while((UART1->USR2 & 0x1) == 0);   /* 等待接收完成      */
245     return UART1->URXD;                /* 返回接收到的数据  */
246 }
247
248 /*
249  * @description   : 防止编译器报错
250  * @param         : 无
251  * @return        : 无
252  */
253 void raise(int sig_nr)
254 {
255
256 }
```

文件 bsp_uart.c 中共有 10 个函数，我们依次来看一下这些函数都是做什么的，第一个函数是 uart_init，这个函数是 UART1 初始化函数，用于初始化 UART1 相关的 IO、并且设置 UART1 的波特率、字长、停止位和校验模式等，初始化完成以后就使能 UART1。第二个函数是 uart_io_init，用于初始化 UART1 所使用的 IO。第三个函数是 uart_setbaudrate，这个函数是从 NXP 官方的 SDK 包里面移植过来的，用于设置波特率。我们只需将要设置的波特率告诉此函数，此函数就会使用逐次逼近方式来计算出寄存器 UART1_UFCR 的 FRDIV 位、寄存器 UART1_UBIR 和寄存器 UART1_UBMR 这三个的值。第四和第五这两个函数为 uart_disable 和 uart_enable，分别是使能和关闭 UART1。第 6 个函数是 uart_softreset，用于软件复位指定的 UART。第七个函数是 putc，用于通过 UART1 发送一个字节的数据。第八个函数是 puts，用于通过 UART1 发送一串数据。第九个函数是 getc，用于通过 UART1 获取一个字节的数据，最后一个函数是 raise，这是一个空函数，防止编译器报错。

最后在 main.c 中输入如下所示内容：

示例代码 21.3.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
```

```
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 13 串口实验
其他      : 本实验我们学习如何使用 I.MX6 的串口，实现串口收发数据，了解
            I.MX6 的串口工作原理。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
**************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8
9  /*
10  * @description    : main 函数
11  * @param          : 无
12  * @return         : 无
13  */
14 int main(void)
15 {
16     unsigned char a=0;
17     unsigned char state = OFF;
18
19     int_init();                /* 初始化中断(一定要最先调用！) */
20     imx6u_clkinit();           /* 初始化系统时钟              */
21     delay_init();              /* 初始化延时                  */
22     clk_enable();              /* 使能所有的时钟              */
23     led_init();                /* 初始化 led                  */
24     beep_init();               /* 初始化 beep                 */
25     uart_init();               /* 初始化串口，波特率 115200    */
26
27     while(1)
28     {
29         puts("请输入 1 个字符:");
30         a=getc();
31         putc(a);               /* 回显功能 */
32         puts("\r\n");
33
34         /* 显示输入的字符 */
35         puts("您输入的字符为:");
```

```
36          putc(a);
37          puts("\r\n\r\n");
38
39          state = !state;
40          led_switch(LED0,state);
41      }
42      return 0;
43 }
```

第 5 行调用函数 uart_init 初始化 UART1，最终在 while 循环里面获取串口接收到的数据，并且将获取到的数据通过串口打印出来。

21.4 编译下载验证

21.4.1 编写 Makefile 和链接脚本

在 Makefile 文件中输入如下内容：

示例代码 21.4.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= uart
3
4  CC             := $(CROSS_COMPILE)gcc
5  LD             := $(CROSS_COMPILE)ld
6  OBJCOPY        := $(CROSS_COMPILE)objcopy
7  OBJDUMP        := $(CROSS_COMPILE)objdump
8
9  LIBPATH        := -lgcc -L /usr/local/arm/gcc-linaro-7.3.1-2018.05-
            x86_64_arm-linux-gnueabihf/lib/gcc/arm-linux-gnueabihf/7.3.1
10
11
12 INCDIRS        :=  imx6ul \
13                    bsp/clk \
14                    bsp/led \
15                    bsp/delay  \
16                    bsp/beep \
17                    bsp/gpio \
18                    bsp/key \
19                    bsp/exit \
20                    bsp/int \
21                    bsp/epittimer \
22                    bsp/keyfilter \
23                    bsp/uart
24
25 SRCDIRS        :=  project \
26                    bsp/clk \
```

```
27                   bsp/led \
28                   bsp/delay \
29                   bsp/beep \
30                   bsp/gpio \
31                   bsp/key \
32                   bsp/exit \
33                   bsp/int \
34                   bsp/epittimer \
35                   bsp/keyfilter \
36                   bsp/uart
37
38
39 INCLUDE        := $(patsubst %, -I %, $(INCDIRS))
40
41 SFILES         := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/*.S))
42 CFILES         := $(foreach dir, $(SRCDIRS), $(wildcard $(dir)/*.c))
43
44 SFILENDIR      := $(notdir  $(SFILES))
45 CFILENDIR      := $(notdir  $(CFILES))
46
47 SOBJS          := $(patsubst %, obj/%, $(SFILENDIR:.S=.o))
48 COBJS          := $(patsubst %, obj/%, $(CFILENDIR:.c=.o))
49 OBJS           := $(SOBJS) $(COBJS)
50
51 VPATH          := $(SRCDIRS)
52
53 .PHONY: clean
54
55 $(TARGET).bin : $(OBJS)
56     $(LD) -Timx6ul.lds -o $(TARGET).elf $^ $(LIBPATH)
57     $(OBJCOPY) -O binary -S $(TARGET).elf $@
58     $(OBJDUMP) -D -m arm $(TARGET).elf > $(TARGET).dis
59
60 $(SOBJS) : obj/%.o : %.S
61     $(CC) -Wall -nostdlib -fno-builtin -c -O2  $(INCLUDE) -o $@ $<
62
63 $(COBJS) : obj/%.o : %.c
64     $(CC) -Wall -nostdlib -fno-builtin -c -O2  $(INCLUDE) -o $@ $<
65
66 clean:
67  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

上述的 Makefile 文件内容和上一章实验的区别不大。将 TARGET 为 uart，在 INCDIRS 和 SRCDIRS 中加入“bsp/uart”。但是，相比上一章中的 Makefile 文件，本章实验的 Makefile 有两处重要的改变：

①、本章 Makefile 文件在链接的时候加入了数学库， 因为在 bsp_uart.c 中有个函数 uart_setbaudrate，在此函数中使用到了除法运算，因此在链接的时候需要将编译器的数学库也链接进来。第 9 行的变量 LIBPATH 就是数学库的目录，在第 56 行链接的时候使用了变量 LIBPATH。

在后面的学习中，我们常常要用到一些第三方库，那么在连接程序的时候就需要指定这些第三方库所在的目录，Makefile 在链接的时候使用选项“-L”来指定库所在的目录，比如“示例代码 21.4.1”中第 9 行的变量 LIBPATH 就是指定了我们所使用的编译器库所在的目录。

②、在第 61 行和 64 行中，加入了选项“-fno-builtin”，否则编译的时候提示“putc”、“puts”这两个函数与内建函数冲突，错误信息如下所示：

```
warning: conflicting types for built-in function ‘putc’
warning: conflicting types for built-in function ‘puts’
```

在编译的时候加入选项“-fno-builtin”表示不使用内建函数，这样我们就可以自己实现 putc 和 puts 这样的函数了。

链接脚本保持不变。

21.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 uart.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload uart.bin /dev/sdd      //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。打开 SourceCRT，点击 File->Quick Connect…，打开快速连接设置界面，设置好相应的串口参数，比如在我的电脑上是 COM8，设置如图 21.4.2.1 所示：

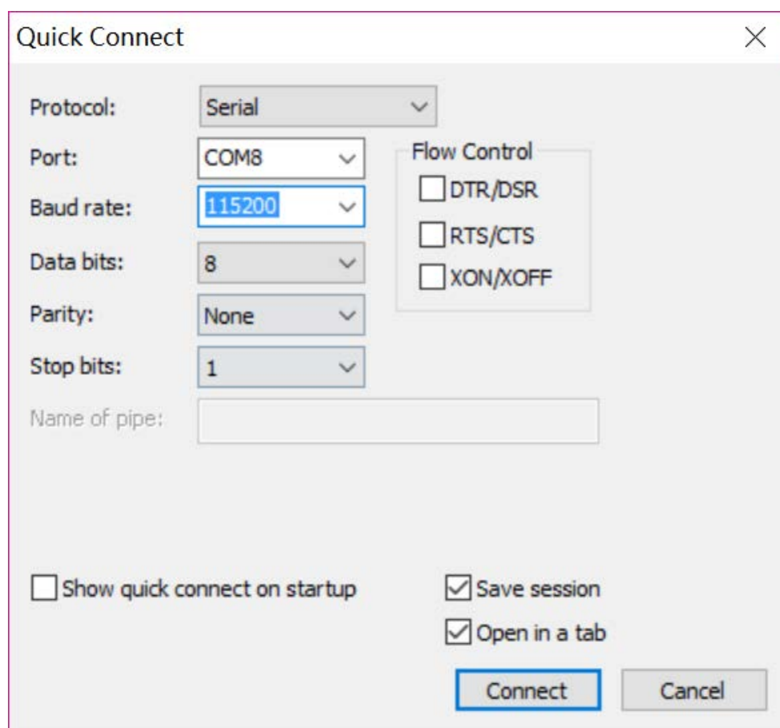


图 21.4.2.1 SecureCRT 串口设置

设置好以后就点击“Connect”就可以了，连接成功以后 SecureCRT 收到来自开发板的数据，但是 SecureCRT 显示可能会是乱码，如图 21.4.2.2 所示：

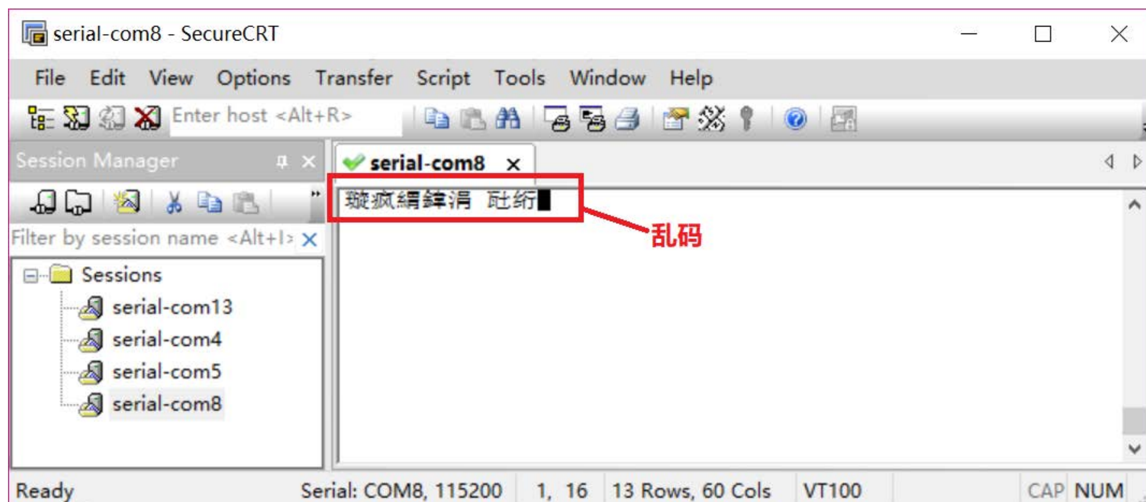


图 21.4.2.2 SecureCRT 显示乱码

这是因为有些设置还没做，点击 Options->Session Options…，打开会话设置窗口，按照图 21.4.2.3 所示设置：

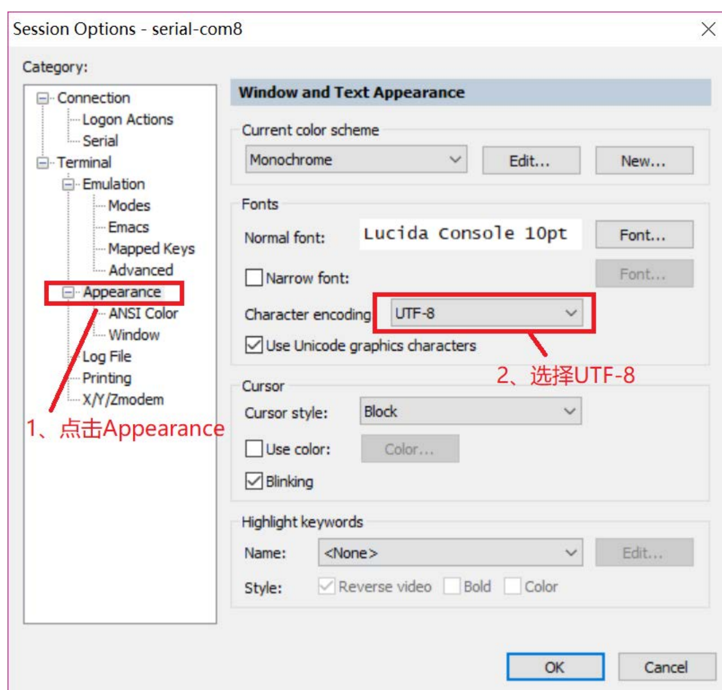


图 21.4.2.3 会话设置

设置好以后点击“OK”按钮就可以了，清屏，然后重新复位一次开发板，此时 SecureCRT 显示就正常了，如图 21.4.2.4 所示：



图 21.4.2.4 显示正常

根据提示输入一个字符，这个输入的字符就会通过串口发送给开发板，开发板接收到字符以后就会通过串口提示你接收到的字符是什么，如图 21.4.2.5 所示：

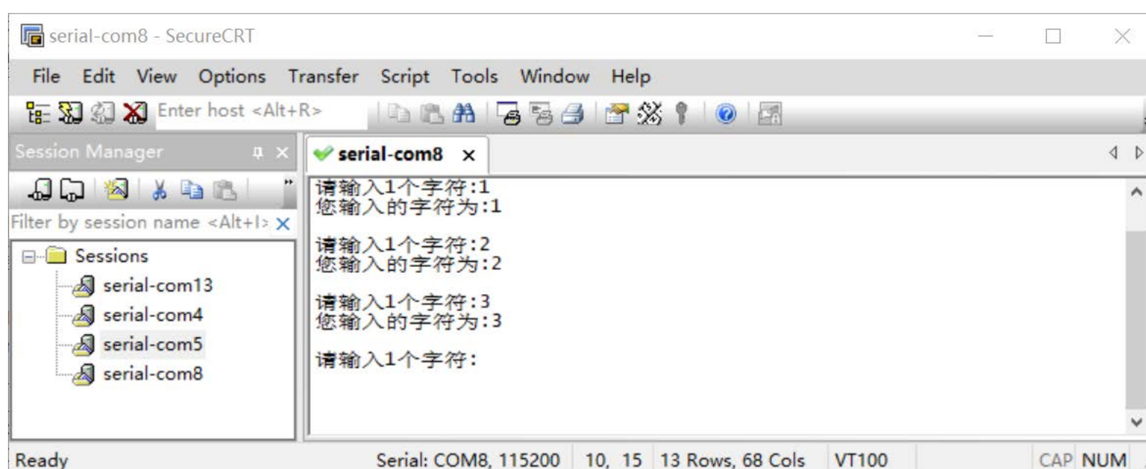


图 21.4.2.5 实验效果

至此，I.MX6U 的串口 1 就工作起来了，以后我们就可以通过串口来调试程序。但是本章只实现了串口最基本的收发功能，如果我们要想使用格式化输出话就不行了，比如最常用的 printf 函数，下一章就讲解如何移植 printf 函数。

第二十二章 串口格式化函数移植实验

上一章实验我们实现了 UART1 基本的数据收发功能，虽然可以用来调试程序，但是功能太单一了，只能输出字符。如果需要输出数字的时候就需要我们自己先将数字转换为字符，非常的不方便。学习 STM32 串口的时候我们都会将 printf 函数映射到串口上，这样就可以使用 printf 函数来完成格式化输出了，使用非常方便。本章我们就来学习如何将 printf 这样的格式化函数移植到 I.MX6U-ALPHA 开发板上。

22.1 串口格式化函数简介

格式化函数说的是 printf、sprintf 和 scanf 这样的函数，分为格式化输入和格式化输出两类函数。学习 C 语言的时候常常通过 printf 函数在屏幕上显示字符串，通过 scanf 函数从键盘获取输入。这样就有了输入和输出了，实现了最基本的人机交互。学习 STM32 的时候会将 printf 映射到串口上，这样即使没有屏幕，也可以通过串口来和开发板进行交互。在 I.MX6U-ALPHA 开发板上也可以使用此方法，将 printf 和 scanf 映射到串口上，这样就可以使用 SecureCRT 作为开发板的终端，完成与开发板的交互。也可以使用 printf 和 sprintf 来实现各种各样的格式化字符串，方便我们后续的开发。串口驱动我们上一章已经编写完成了，而且实现了最基本的字节收发，本章我们就通过移植网上别人已经做好的文件来实现格式化函数。

22.2 硬件原理分析

本章所需的硬件和上一章相同。

22.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 14_printf。

本章实验所需要移植的源码已经放到了开发板光盘中，路径为：1、例程源码->5、模块驱动源码->2、格式化函数源码->stdio，文件夹 stdio 里面的文件就是我们要移植的源码文件。本章实验在上一章例程的基础上完成，将 stdio 文件夹复制到实验工程根目录中，如图 22.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/14_printf$ ls -a
.  ..  bsp  imx6ul  imx6ul.lds  imxdownload  Makefile  obj  project  stdio  .vscode
zuozhongkai@ubuntu:~/linux/14_printf$
```

图 22.3.1 添加实验源码

stdio 里面有两个文件夹：include 和 lib，这两个文件夹里面的内容如图 22.3.2 所示：

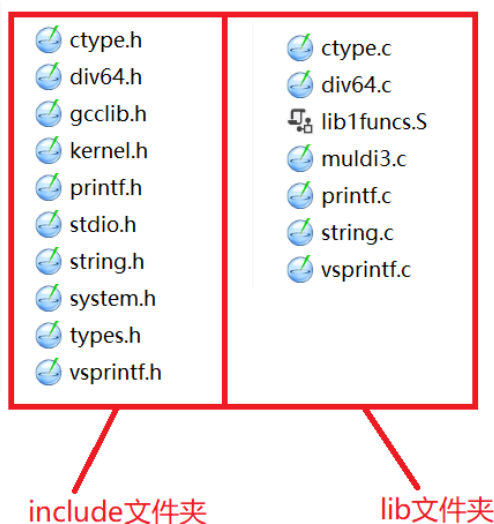


图 22.3.2 stdio 所有源码文件

图 22.3.2 就是 stdio 里面的所有文件，stdio 里面的文件其实是从 uboot 里面移植过来的。后面学习 uboot 以后大家有兴趣的话可以自行从 uboot 源码里面“扣”出相应的文件，完成格式化函数的移植。这里要注意一点，stdio 中并没有实现完全版的格式化函数，比如 printf 函数并不支持浮点数，但是基本够我们使用了。

移植好以后就要测试相应的函数工作是否正常，我们使用 scanf 函数等待键盘输入两个整数，然后将两个整数进行相加并使用 printf 函数输出结果。在 main.c 里面输入如下内容：

示例代码 22.3.1 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : main.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 14 串口 print 实验
其他      : 本实验在串口上移植 printf，实现 printf 函数功能，方便以后的
            程序调试。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "stdio.h"
9  
10 /*
11  * @description    : main 函数
12  * @param          : 无
13  * @return         : 无
14  */
15 int main(void)
16 {
17     unsigned char state = OFF;
18     int a , b;
19 
20     int_init();          /* 初始化中断(一定要最先调用！) */
21     imx6u_clkinit();     /* 初始化系统时钟              */
22     delay_init();        /* 初始化延时                  */
23     clk_enable();        /* 使能所有的时钟              */
24     led_init();          /* 初始化 led                  */
25     beep_init();         /* 初始化 beep                 */
26     uart_init();         /* 初始化串口，波特率 115200    */
27 
28     while(1)
29     {
30         printf("输入两个整数，使用空格隔开:");
31         scanf("%d %d", &a, &b);            /* 输入两个整数 */
```

```
32          printf("\r\n 数据%d + %d = %d\r\n\r\n", a, b, a+b);/* 输出和 */
33
34          state = !state;
35          led_switch(LED0,state);
36      }
37
38      return 0;
39  }
```

第 30 行使用 printf 函数输出一段提示信息，第 31 行使用函数 scanf 等待键盘输入两个整数。第 32 行使用 printf 函数输出两个整数的和。程序很简单，但是可以验证 printf 和 scanf 这两个函数是否正常工作。

22.4 编译下载验证

22.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 printf，在 INCDIRS 中加入"stdio/include"，在 SRCDIRS 中加入"stdio/lib"，修改后的 Makefile 如下：

示例代码 22.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= printf
3
4  /* 省略掉其它代码...... */
5
6  INCDIRS        := imx6ul \
7                    stdio/include \
8                    bsp/clk \
9                    bsp/led \
10                   bsp/delay  \
11                   bsp/beep \
12                   bsp/gpio \
13                   bsp/key \
14                   bsp/exit \
15                   bsp/int \
16                   bsp/epittimer \
17                   bsp/keyfilter \
18                   bsp/uart
19
20 SRCDIRS        := project \
21                   stdio/lib \
22                   bsp/clk \
23                   bsp/led \
24                   bsp/delay \
25                   bsp/beep \
```

```
26                     bsp/gpio \
27                     bsp/key \
28                     bsp/exit \
29                     bsp/int \
30                     bsp/epittimer \
31                     bsp/keyfilter \
32                     bsp/uart
33
34 /* 省略掉其它代码...... */
35
36 $(COBJS) : obj/%.o : %.c
37  $(CC) -Wall -Wa,-mimplicit-it=thumb -nostdlib -fno-builtin -c -O2 
$(INCLUDE) -o $@ $<
38
39 clean:
40  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“printf”，也就是目标名称为“printf”。

第 7 行在变量 INCDIRS 中添加 stdio 相关头文件(.h)路径。

第 28 行在变量 SRCDIRS 中添加 stdio 相关文件(.c)路径。

第 37 行在编译 C 文件的时候添加了选项“-Wa,-mimplicit-it=thumb”，否则的话会有如下类似的错误提示：

```
thumb conditional instruction should be in IT block -- `addcs r5,r5,#65536'
```

链接脚本保持不变。

22.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 printf.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                  //给予 imxdownload 可执行权限，一次即可
./imxdownload printf.bin /dev/sdd      //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，打开 SourceCRT，设置好连接，然后复位开发板。SourceCRT 显示如图 22.4.2.1 所示：

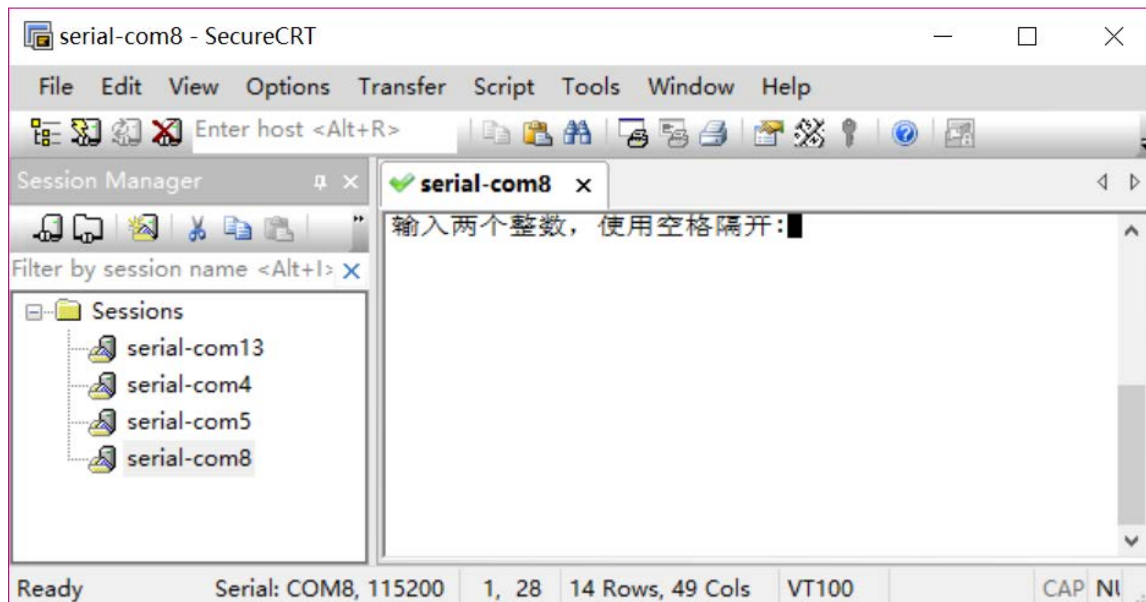


图 22.4.2.1 SourceCRT 默认显示界面

根据图 22.4.2.1 所示的提示，输入两个整数，使用空格隔开，输入完成以后按下“回车键”，结果如图 22.4.2.2 所示：

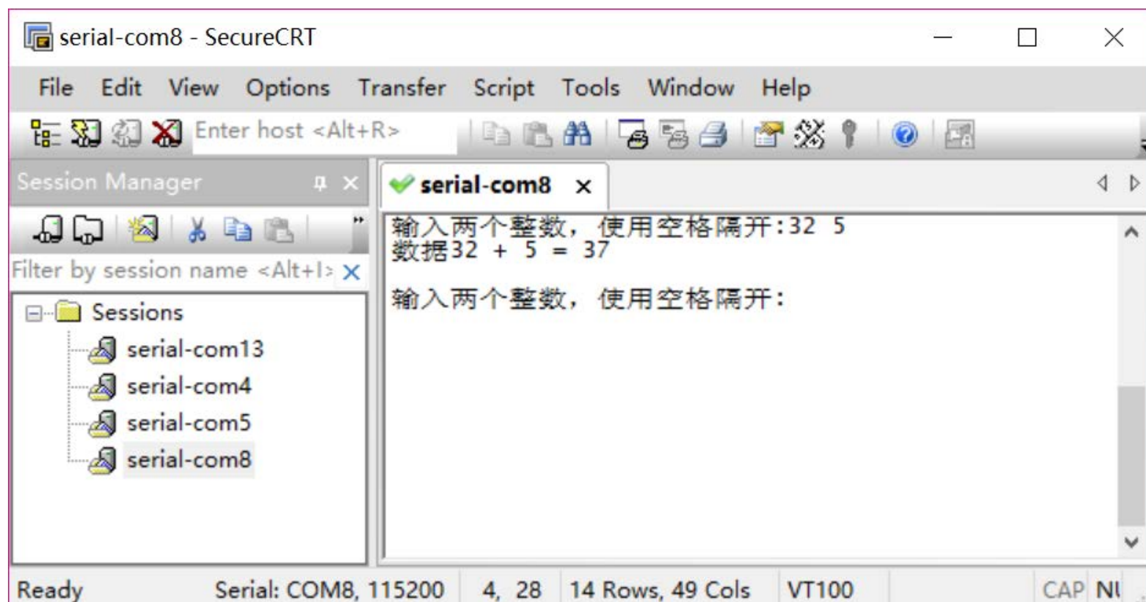


图 22.4.2.2 计算输入结果显示

从图 22.4.2.2 可以看出，输入了 32 和 5，这两个整数，然后计算出 32+5=37。计算和显示都正确，说明格式化函数移植成功，以后我们就可以使用 printf 来调试程序了。

第二十三章 DDR3 实验

I.MX6U-ALPHA 开发板上带有一个 256MB/512MB 的 DDR3 内存芯片，一般 Cortex-A 芯片自带的 RAM 很小，比如 I.MX6U 只有 128KB 的 OCRAM。如果要运行 Linux 的话完全不够用的，所以必须要外接一片 RAM 芯片，I.MX6U 支持 LPDDR2、LPDDR3/DDR3，I.MX6U-ALPHA 开发板上选择的是 DDR3，本章就来学习如何驱动 I.MX6U-ALPHA 开发板上的这片 DDR3。

23.1 DDR3 内存简介

在正式学习 DDR3 内存之前，我们要先了解一下 DDR 内存的发展历史，通过对比 SRAM、SDRAM、DDR、DDDR2 和 DDR3 的区别，有助于我们更加深入的理解什么是 DDR。在看 DDR 之前我们先来了解一个概念，那就是什么叫做 RAM?

23.1.1 何为 RAM 和 ROM?

相信大家在购买手机、电脑等电子设备的时候，通常都会听到 RAM、ROM、硬盘等概念，很多人都是一头雾水的。普通用户区分不清楚 RAM、ROM 到可以理解，但是作为一个嵌入式 Linux 开发者，要是不清楚什么是 RAM、什么是 ROM 就绝对不行！RAM 和 ROM 专业的解释如下：

RAM：随机存储器，可以随时进行读写操作，速度很快，掉电以后数据会丢失。比如内存条、SRAM、SDRAM、DDR 等都是 RAM。RAM 一般用来保存程序数据、中间结果，比如我们在程序中定义了一个变量 a，然后对这个 a 进行读写操作，示例代码如下：

示例代码 23.1.1.1 RAM 中的变量

```
1 int a;
2 a = 10;
```

a 是一个变量，我们需要很方便的对这个变量进行读写操作，方法就是直接“a”进行读写操作，不需要在乎具体的读写过程。我们可以随意的对 RAM 中任何地址的数据进行读写操作，非常方便。

ROM：只读存储器，笔者认为目前“只读存储器”这个定义不准确。比如我们买手机，通常会告诉你这个手机是 4+64 或 6+128 配置，说的就是 RAM 为 4GB 或 6GB，ROM 为 64G 或 128GB。但是这个 ROM 是 Flash，比如 EMMC 或 UFS 存储器，因为历史原因，很多人还是将 Flash 叫做 ROM。但是 EMMC 和 UFS，甚至是 NAND Flash，这些都是可以进行写操作的！只是写起来比较麻烦，要先发送要先进行擦除，然后在发送要写的地址或扇区，最后才是要写入的数据，学习过 STM32，使用过 WM25QXX 系列的 SPI Flash 的同学应该深有体会。可以看出，相比于 RAM，向 ROM 或者 Flash 写入数据要复杂很多，因此意味着速度就会变慢(相比 RAM)，但是 ROM 和 Flash 可以将容量做的很大，而且掉电以后数据不会丢失，适合用来存储资料，比如音乐、图片、视频等信息。

综上所述，RAM 速度快，可以直接和 CPU 进行通信，但是掉电以后数据会丢失，容量不容易做大(和同价格的 Flash 相比)。ROM(目前来说，更适合叫做 Flash)速度虽然慢，但是容量大、适合存储数据。对于正点原子的 I.MX6U-ALPHA 开发板而言，256MB/512MB 的 DDR3 就是 RAM，而 512MB NANF Flash 或 8GB EMMC 就是 ROM。

23.1.2 SRAM 简介

为什么要讲 SRAM 呢？因为大多数的朋友最先接触 RAM 芯片都是从 SRAM 开始的，因为大量的 STM32 单片机开发板都使用到了 SRAM，比如 F103、F407 等，基本都会外扩一个 512KB 或 1MB 的 SRAM 的，因为 STM32F103/F407 内部 RAM 比较小，在一些比较耗费内存的应用中会出现内存捉紧的情况，比如 emWin 做 UI 界面。我们简单回顾一下 SRAM，如果想要详细的了解 SRAM 请阅读正点原子 STM32F103 战舰开发板的开发指南。

SRAM 的全称叫做 Static Random-Access Memory，也就是静态随机存储器，这里的“静态”说的就是只要 SRAM 上电，那么 SRAM 里面的数据就会一直保存着，直到 SRAM 掉电。对于 RAM 而言需要可以随机的读取任意一个地址空间内的数据，因此采用了地址线和数据线分离

的方式，这里就以 STM32F103/F407 开发板常用的 IS62WV51216 这颗 SRAM 芯片为例简单的讲解一下 SRAM，这是一颗 16 位宽(数据位为 16 位)、1MB 大小的 SRAM，芯片框图如图 23.1.2.1 所示：

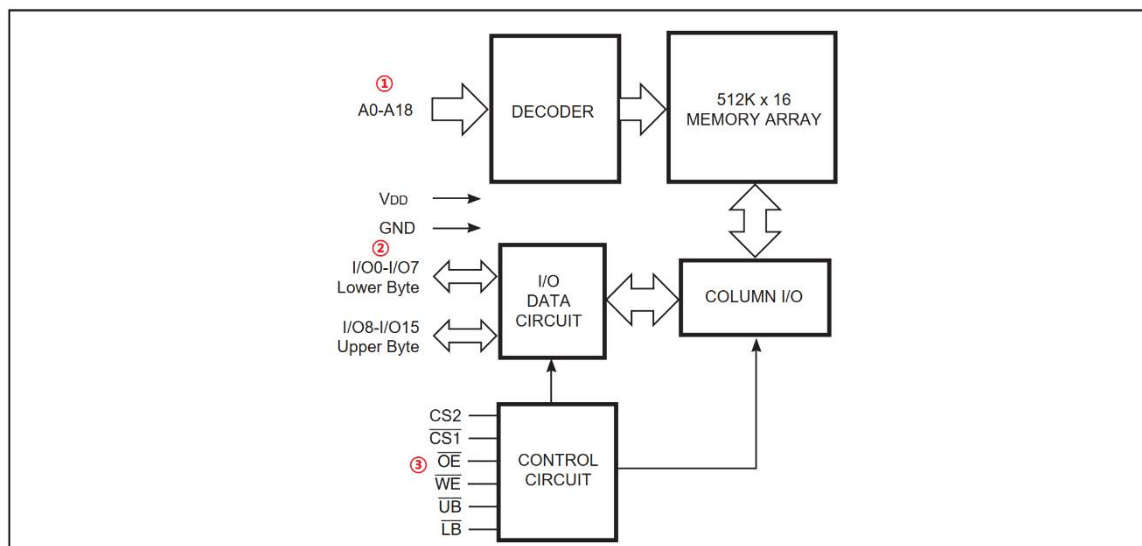


图 23.1.2.1 IS62WV51216 框图

图 23.1.2.1 主要分为三部分，我们依次来看一下这三部分：

①、地址线

这部分是地址线，一共 A0~A18，也就是 19 根地址线，因此可访问的地址大小就是 2^19=524288=512KB。不是说 IS62WV51216 是个 1MB 的 SRAM 吗？为什么地址空间只有 512KB？前面我们说了 IS62WV51216 是 16 位宽的，也就是一次访问 2 个字节，因此需要对 512KB 进行乘 2 处理，得到 512KB*2=1MB。位宽的话一般有 8 位/16 位/32 位，根据实际需求选择即可，一般都是根据处理器的 SRAM 控制器位宽来选择 SRAM 位宽。

②、数据线

这部分是 SRAM 的数据线，根据 SRAM 位宽的不同，数据线的数量要不同，8 位宽就有 8 根数据线，16 位宽就有 16 根数据线，32 位宽就有 32 根数据线。IS62WV51216 是一个 16 位宽的 SRAM，因此就有 16 根数据线，一次访问可以访问 16bit 的数据，也就是 2 个字节。因此就有高字节和低字节数据之分，其中 IO0~IO7 是低字节数据，IO8~IO15 是高字节数据。

③、控制线

SRAM 要工作还需要一堆的控制线，CS2 和 CS1 是片选信号，低电平有效，在一个系统中可能会有多片 SRAM(目的是为了扩展 SRAM 大小或位宽)，这个时候就需要 CS 信号来选择当前使用哪片 SRAM。另外，有的 SRAM 内部其实是由两片 SRAM 拼接起来的，因此就会提供两个片选信号。

OE 是输出使能信号，低电平有效，也就是主控从 SRAM 读取数据。

WE 是写使能信号，低电平有效，也就是主控向 SRAM 写数据。

UB 和 LB 信号，前面我们已经说了，IS62WV51216 是个 16 位宽的 SRAM，分为高字节和低字节，那么如何来控制读取高字节数据还是低字节数据呢？这个就是 UB 和 LB 这两个控制线的作用，这两根控制线都是低电平有效。UB 为低电平的话表示访问高字节，LB 为低电平的话表示访问低字节。关于 IS62WV51216 的简单原理就讲解到这里。

那么 SRAM 有什么缺点没有？那必须有的啊，要不然就不可能有本章教程了，SRAM 最大

的缺点就是成本高！价格高，大家可以在淘宝上搜索一下 IS62WV51216 这个仅仅只有 1MB 大小的 SRAM 售价为多少，大概为 5,6 块钱。大家在搜索一下 32MB 的 SDRAM 多钱，以华邦的 W9825G6KH 为例，大概 4,5 块钱，可以看出 SDRAM 比 SRAM 容量大，但是价格更低。SRAM 突出的特点就是无需刷新(SDRAM 需要刷新，后面会讲解)，读写速度快！所以 SRAM 通常作为 SOC 的内部 RAM 使用或 Cache 使用，比如 STM32 内存的 RAM 或 I.MX6U 内部的 OCRAM 都是 SRAM。

23.1.3 SDRAM 简介

前面给大家简单讲解了 SRAM，可以看出 SRAM 最大的缺点就是价格高、容量小！但是应用对于内存的需求越来越高，必须提供大内存解决方案。为此半导体厂商想了很多办法，提出了很多解决方法，最终 SDRAM 营运而生，得到推广。SDRAM 全称是 Synchronous Dynamic Random Access Memory，翻译过来就是同步动态随机存储器，“同步”的意思是 SDRAM 工作需要时钟线，“动态”的意思是 SDRAM 中的数据需要不断的刷新来保证数据不会丢失，“随机”的意思就是可以读写任意地址的数据。

与 SRAM 相比，SDRAM 集成度高、功耗低、成本低、适合做大容量存储，但是需要定时刷新来保证数据不会丢失。因此 SDRAM 适合用来做内存条，SRAM 适合做高速缓存或 MCU 内部的 RAM。SDRAM 目前已经发展到了第四代，分别为：SDRAM、DDR SDRAM、DDR2 SDRAM、DDR3 SDRAM、DDR4 SDRAM。STM32F429/F767/H743 等芯片支持 SDRAM，学过 STM32F429/F767/H743 的朋友应该知道 SDRAM，这里我们就以 STM32 开发板最常用的华邦 W9825G6KH 为例，W9825G6KH 是一款 16 位宽(数据位为 16 位)、32MB 的 SDRAM、速度一般为 133MHz、166MHz 或 200MHz。W9825G6KH 框图如图 23.1.3.1 所示：

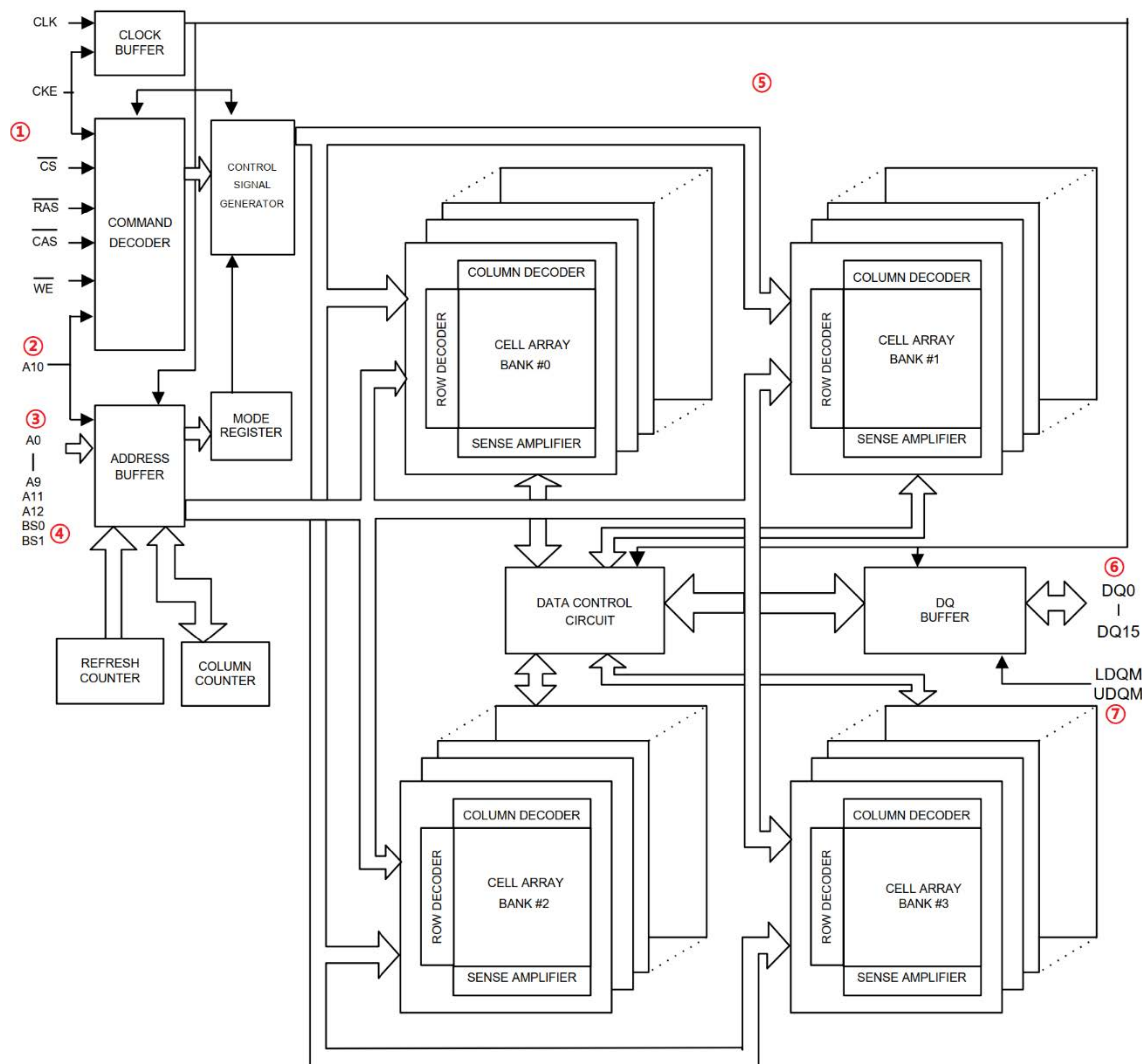


图 23.1.3.1 W9825G6KH 框图

①、控制线

SDRAM 也需要很多控制线，我们依次来看一下：

CLK：时钟线，SDRAM 是同步动态随机存储器，“同步”的意思就是时钟，因此需要一根额外的时钟线，这是和 SRAM 最大的不同，SRAM 没有时钟线。

CKE：时钟使能信号线，SRAM 没有 CKE 信号。

CS：片选信号，这个和 SRAM 一样，都有片选信号。

RAS：行选通信号，低电平有效，SDRAM 和 SRAM 的寻址方式不同，SDRAM 按照行、列来确定某个具体的存储区域。因此就有行地址和列地址之分，行地址和列地址共同复用同一组地址线，要访问某一个地址区域，必须要发送行地址和列地址，指定要访问哪一行？哪一列？RAS 是行选通信号，表示要发送行地址，行地址和列地址访问方式如图 23.1.3.2 所示：

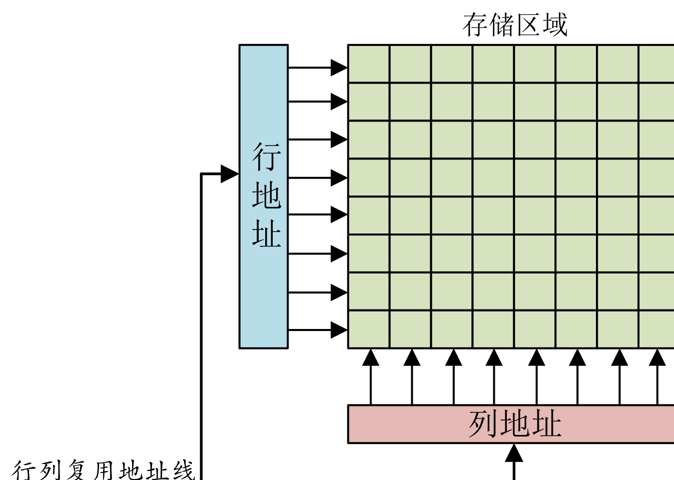


图 23.1.3.2 SDRAM 行列寻址方式

CAS： 列选通信号，和 RAS 类似，低电平有效，选中以后就可以发送列地址了。

WE： 写使能信号，低电平有效。

②、A10 地址线

A10 是地址线，那么这里为什么要单独将 A10 地址线给提出来呢？因为 A10 地址线还有另外一个作用，A10 还控制着 Auto-precharge，也就是预充电。这里又提到了预充电的概念，SDRAM 芯片内部会分为多个 BANK，关于 BANK 我们稍后会讲解。SDRAM 在读写完成以后，如果要对同一个 BANK 中的另一行进行寻址操作就必须将原来有效的行关闭，然后发送新的行/列地址，关闭现在工作的行，准备打开新行的操作就叫做预充电。一般 SDSRAM 都支持自动预充电的功能。

③、地址线

对于 W9825G6KH 来说一共有 A0~A12，共 13 根地址线，但是我们前面说了 SDRAM 寻址是按照行地址和列地址来访问的，因此这 A0~A12 包含了行地址和列地址。不同的 SDRAM 芯片，根据其位宽、容量等的不同，行列地址数是不同的，这个在 SDRAM 的数据手册里面会也清楚的。比如 W9825G6KH 的 A0~A8 是列地址，一共 9 位列地址，A0~A12 是行地址，一共 13 位，因此可寻址范围为：2^9*2^13=4194304B=4MB，W9825G6KH 为 16 位宽(2 个字节)，因此还需要对 4MB 进行乘 2 处理，得到 4*2=8MB，但是 W9825G6KH 是一个 32MB 的 SDRAM 啊，为什么算出来只有 8MB，仅仅为实际容量的 1/4。不要急，这个就是我们接下来要讲的 BANK，8MB 只是一个 BANK 的容量，W9825G6KH 一共有 4 个 BANK。

④、BANK 选择线

BS0 和 BS1 是 BANK 选择信号线，在一片 SDRAM 中因为技术、成本等原因，不可能做一个全容量的 BANK。而且，因为 SDRAM 的工作原理，单一的 BANK 会带来严重的寻址冲突，减低内存访问效率。为此，人们在一片 SDRAM 中分割出多块 BANK，一般都是 2 的次方，比如 2，4，8 等。图 23.1.1.2 中的⑤就是 W9825G6KH 就是 4 个 BANK 示意图，每个 SDRAM 数据手册里面都会写清楚自己是几 BANK。前面我们已经计算出来了一个 BANK 的大小为 8MB，那么四个 BANK 的总容量就是 8MB*4=32MB。

既然有 4 个 BANK，那么在访问的时候就需要告诉 SDRAM，我们现在需要访问哪个 BANK，BS0 和 BS1 就是为此而生的，4 个 BANK 刚好 2 根线，如果是 8 个 BANK 的话就需要三根线，也就是 BS0~BS2。BS0、BS1 这两个线也是 SRAM 所没有的。

⑤、BANK 区域

关于 BANK 的概念前面已经讲过了，这部分就是 W9825G6KH 的 4 个 BANK 区域。这个概念也是 SRAM 所没有的。

⑥、数据线

W9825G6KH 是 16 位宽的 SDRAM，因此有 16 根数据线，DQ0~DQ15，不同的位宽其数据线数量不同，这个和 SRAM 是一样的。

⑦、高低字节选择

W9825G6KH 是一个 16 位的 SDRAM，因此就分为低字节数据和高字节数据，LDQM 和 UDQM 就是低字节和高字节选择信号，这个也和 SRAM 一样。

23.1.4 DDR 简介

终于到了 DDR 内存了，DDR 内存是 SDRAM 的升级版本，SDRAM 分为 SDR SDRAM、DDR SDRAM、DDR2 SDRAM、DDR3 SDRAM、DDR4 SDRAM。可以看出 DDR 本质上还是 SDRAM，只是随着技术的不断发展，DDR 也在不断的更新换代。先来看一下 DDR，也就是 DDR1，人们对于速度的追求是永无止境的，当发现 SDRAM 的速度不够快的时候人们就在思考如何提高 SDRAM 的速度，DDR SDRAM 由此诞生。

DDR 全称是 Double Data Rate SDRAM，也就是双倍速率 SDRAM，看名字就知道 DDR 的速率(数据传输速率)比 SDRAM 高一倍！这 1 倍的速度不是简简单单的将 CLK 提高 1 倍，SDRAM 在一个 CLK 周期传输一次数据，DDR 在一个 CLK 周期传输两次数据，也就是在上升沿和下降沿各传输一次数据，这个概念叫做预取(prefetch)，相当于 DDR 的预取为 2bit，因此 DDR 的速度直接加倍！比如 SDRAM 速度一般是 133~200MHz，对应的传输速度就是 133~200MT/s，在描述 DDR 速度的时候一般都使用 MT/s，也就是每秒多少兆次数据传输。133MT/S 就是每秒 133M 次数据传输，MT/s 描述的是单位时间内传输速率。同样 133~200MHz 的频率，DDR 的传输速度就变为了 266~400MT/S，所以大家常说的 DDR266、DDR400 就是这么来的。

DDR2 的 IO 时钟是 DDR 的 2 倍，因此 DDR 内核时钟依旧是 133~200MHz 的时候，总线速度就是 266~400MHz。而且 DDR2 在 DDR 基础上进一步增加预取(prefetch)，增加到了 4bit，相当于比 DDR 多读取一倍的数据，因此 DDR2 的数据传输速率就是 533~800MT/s，这个也就是大家常说的 DDR2 533、DDR2 800。当然了，DDR2 还有其他速度，这里只是说最常见的几种。

DDR3 在 DDR2 的基础上将预取(prefetch)提高到 8bit，因此又获得了比 DDR2 高一倍的传输速率，因此在总线时钟同样为 266~400MHz 的情况下，DDR3 的传输速率就是 1066~1600MT/S。I.MX6U 的 MMDC 外设用于连接 DDR，支持 LPDDR2、DDR3、DDR3L，最高支持 16 位数据位宽。总线速度为 400MHz(实际是 396MHz)，数据传输速率最大为 800MT/S。这里我们讲一下 LPDDR3、DDR3 和 DDR3L 的区别，这三个都是 DDR3，但是区别主要在于工作电压，LPDDR3 叫做低功耗 DDR3，工作电压为 1.2V。DDR3 叫做标压 DDR3，工作电压为 1.5V，一般台式内存条都是 DDR3。DDR3L 是低压 DDR3，工作电压为 1.35V，一般手机、嵌入式、笔记本等都使用 DDR3L。

正点原子的 I.MX6U-ALPHA 开发板上接了一个 256MB/512MB 的 DDR3L，16 位宽，型号为 NT5CC128M16JR/MT5CC256M16EP，nanya 公司出品的，分为对应 256MB 和 512MB 容量。EMMC 核心板上用的 512MB 容量的 DDR3L，NAND 核心板上用的 256MB 容量的 DDR3L。本

讲解我们就以 EMMC 核心板上使用的 NT5CC256M16EP-EK 为例讲解一下 DDR3。可以到 nanya 官网去查找一下此型号，信息如图 23.1.4.1 所示：

NT5CC256M16EP-EK

Specifications

Density	Config	Voltage	Package
4Gb	x16	1.35V	BGA
Ball Number	**Speed**	**Temperature**	**Grade**
96-ball	1866Mbps	0C~95C	Commercial
Availability			
MP			

图 23.1.4.1 NT5CC256M16EP-EK 信息

从图 23.1.4.1 可以看出，NT5CC256M16EP-EK 是一款容量为 4Gb，也就是 512MB 大小、16 位宽、1.35V、传输速率为 1866MT/S 的 DDR3L 芯片。NT5CC256M16EP-EK 的数据手册没有在 nanya 官网找到，但是找到了 NT5CC256M16ER-EK 数据手册，在官网上没有看出这两个有什么区别，因此我们就直接用 NT5CC256M16ER-EK 的数据手册。数据手册已经放到了开发板光盘中，路径为：6、硬件资料-》1、芯片资料-》NT5CC256M16EP-EK.pdf。但是数据手册并没有给出 DDR3L 对的结构框图，这里我就直接用镁光 MT41K256M16 数据手册里面的结构框图了，都是一样的，DDR3L 结构框图如图 23.1.4.2 所示：

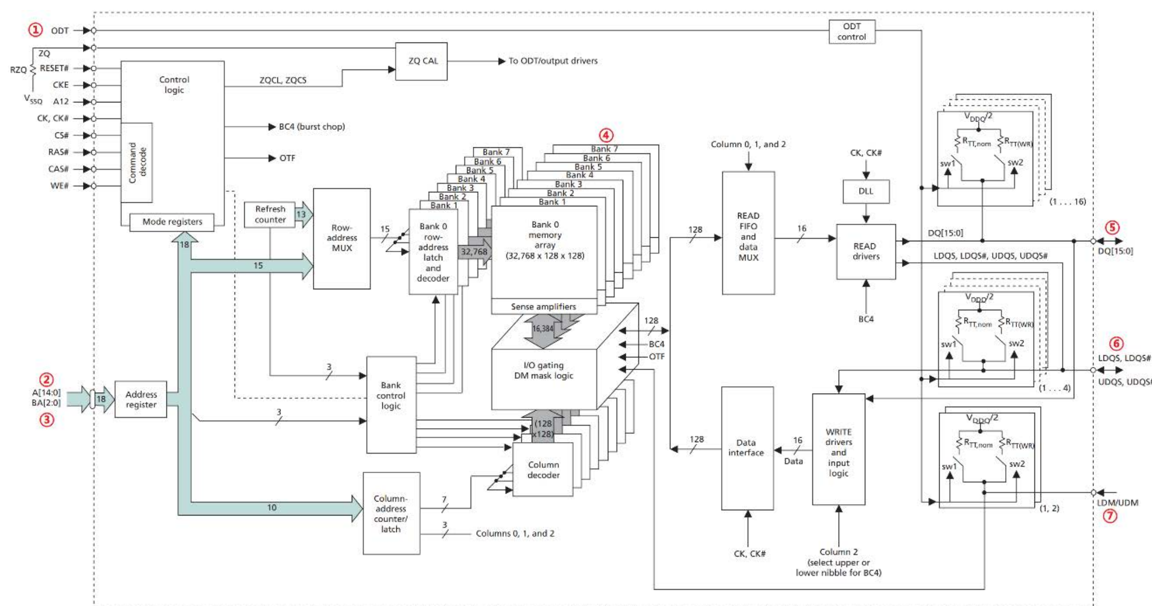


图 23.1.4.2 DDR3L 结构框图

从图 23.1.4.2 可以看出，DDR3L 和 SDRAM 对的结构框图很类似，但是还是有点区别。

①、控制线

ODT：片上终端使能，ODT 使能和禁止片内终端电阻。

ZQ：输出驱动校准的外部参考引脚，此引脚应该外接一个 240 欧的电阻到 VSSQ 上，一般就是直接接地了。

RESET：复位引脚，低电平有效。

CKE：时钟使能引脚。

A12：A12 是地址引脚，但是有也有另外一个功能，因此也叫做 BC 引脚，A12 会在 READ 和 WRITE 命令期间被采样，以决定 burst chop 是否会被执行。

CK 和 CK#：时钟信号，DDR3 的时钟线是差分时钟线，所有的控制和地址信号都会在 CK 对的上升沿和 CK#的下降沿交叉处被采集。

CS#：片选信号，低电平有效。

RAS#、CAS#和 WE#：行选通信号、列选通信号和写使能信号。

②、地址线

A[14:0]为地址线，A0~A14，一共 15 根地址线，根据 NT5CC256M16ER-EK 的数据手册可知，列地址为 A0~A9，共 10 根，行地址为 A0~A14，共 15 根，因此一个 BANK 的大小就是 2^10*2^15*2=32MB*2=64MB，根据图 23.1.4.2 可知一共有 8 个 BANK，因此 DDR3L 的容量就是 64*8=512MB。

③、BANK 选择线

一片 DDR3 有 8 个 BANK，因此需要 3 个线才能实现 8 个 BANK 的选择，BA0~BA2 就是用于完成 BANK 选择的。

④、BANK 区域

DDR3 一般都是 8 个 BANK 区域。

⑤、数据线

因为是 16 位宽的，因此有 16 根数据线，分别为 DQ0~DQ15。

⑥、数据选通引脚

DQS 和 DQS#是数据选通引脚，为差分信号，读的时候是输出，写的时候是输入。LDQS(有的叫做 DQSL)和 LDQS#(有的叫做 DQSL#)对应低字节，也就是 DQ0~7，UDQS(有的叫做 DQSU)和 UDQS#(有的叫做 DQSU#)，对应高字节，也就是 DQ8~15。

⑦、数据输入屏蔽引脚

DM 是写数据收入屏蔽引脚。

关于 DDR3L 的框图就讲解到这里，想要详细的了解 DDR3 的组成，请阅读相应对的数据手册。

23.2 DDR3 关键时间参数

大家在购买 DDR3 内存的时候通常会重点观察几个常用的时间参数：

1、传输速率

比如 1066MT/S、1600MT/S、1866MT/S 等，这个是首要考虑的，因为这个决定了 DDR3 内存的最高传输速率。

2、tRCD 参数

tRCD 全称是 RAS-to-CAS Delay，也就是行寻址到列寻址之间的延迟。DDR 的寻址流程是先指定 BANK 地址，然后在指定行地址，最后指定列地址确定最终要寻址的单元。BANK 地址和行地址是同时发出的，这个命令叫做“行激活”(Row Active)。行激活以后就发送列地址和具体的操作命令(读还是写)，这两个是同时发出的，因此一般也用“读/写命令”表示列寻址。在行有效(行激活)到读写命令发出的这段时间间隔叫做 tRCD，如图 23.2.1 所示：

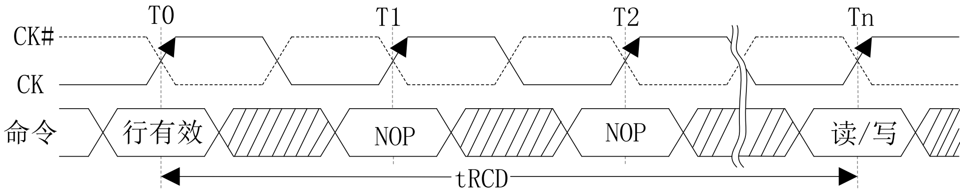


图 23.2.1 tRCD

一般 DDR3 数据手册中都会给出 tRCD 的时间值，比如正点原子所使用的 NT5CC256M16EP-EK 这个 DDR3，tRCD 参数如图 23.2.1 所示：

Speed Bins	DDR3(L)-1866 13-13-13		Unit	Note
Parameter	Min	Max		
tAA	13.91	20.0	ns	
tRCD	13.91	-	ns	
tRP	13.91	-	ns	
tRAS	34.0	9xtREFI	ns	
tRC	47.91	-	ns	

图 23.2.2 tRCD 时间参数

从图 23.2.2 可以看出，tRCD 为 13.91ns，这个我们在初始化 DDR3 的时候需要配置。有时候大家也会看到"13-13-13"之类的参数，这个是用来描述 CL-tRCD-TRP 的，如图 23.2.3 所示：

Organization	Part Number	Package	Speed		
			Clock (MHz)	Data Rate (Mb/s)	CL-TRCD-TRP
DDR3(L) Commercial Grade					
512M x 8	NT5CC512M8EQ-DIB	78-Ball	800	DDR3L-1600 [1]	11-11-11
	NT5CC512M8EQ-DI		800	DDR3L-1600 [1]	11-11-11
	NT5CB512M8EQ-DI		800	DDR3-1600	11-11-11
	NT5CC512M8EQ-EK		933	DDR3L-1866 [1]	13-13-13
	NT5CB512M8EQ-EK		933	DDR3-1866	13-13-13
	NT5CB512M8EQ-FL		1066	DDR3-2133	14-14-14
256M x 16	NT5CC256M16ER-DIB	96-Ball	800	DDR3L-1600 [1]	11-11-11
	NT5CC256M16ER-DI		800	DDR3L-1600 [1]	11-11-11
	NT5CB256M16ER-DI		800	DDR3-1600	11-11-11
	NT5CC256M16ER-EK		933	DDR3L-1866 [1]	13-13-13
	NT5CB256M16ER-EK		933	DDR3-1866	13-13-13
	NT5CB256M16ER-FL		1066	DDR3-2133	14-14-14

图 23.2.3 CL-TRCD-TRP 时间参数

从图 23.2.2 可以看出，NT5CC256M16ER-EK 这个 DDR3 的 CL-TRCD-TRP 时间参数为"13-13-13"。因此 tRCD=13，这里的 13 不是 ns 数，而是 CLK 时间数，表示 13 个 CLK 周期。

3、CL 参数

当列地址发出以后就会触发数据传输，但是从数据从存储单元到内存芯片 IO 接口上还需要一段时间，这段时间就是非常著名的 CL(CAS Latency)，也就是列地址选通潜伏期，如图 23.2.4 所示：

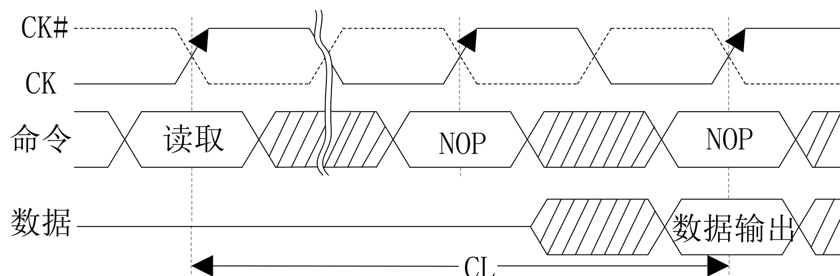


图 23.2.4 CL

CL 参数一般在 DDR3 的数据手册中可以找到，比如 NT5CC256M16EP-EK 的 CL 值就是 13 个时钟周期，一般 tRCD 和 CL 大小一样。

4、AL 参数

在 DDR 的发展中，提出了一个前置 CAS 的概念，目的是为了解决 DDR 中的指令冲突，它允许 CAS 信号紧随着 RAS 发送，相当于将 DDR 中的 CAS 前置了。但是读/写操作并没有因此提前，依旧要保证足够的延迟/潜伏期，为此引入了 AL(Additive Latency)，单位也是时钟周期数。AL+CL 组成了 RL(Read Latency)，从 DDR2 开始还引入了写潜伏期 WL(Write Latency)，WL 表示写命令发出以后到第一笔数据写入的潜伏期。引入 AL 以后的读时序如图 23.2.5 所示：

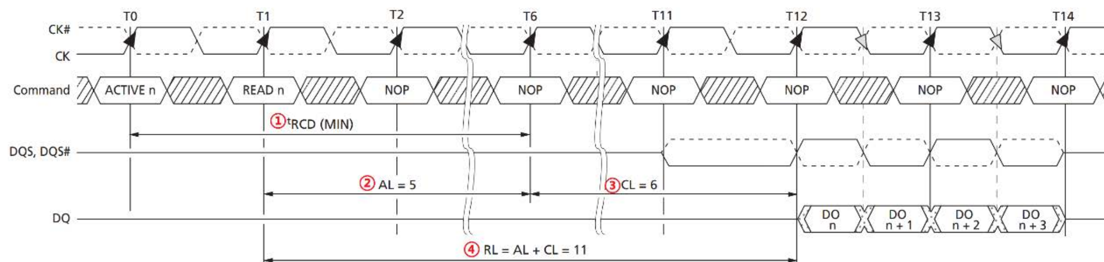


图 23.2.5 加入 AL 后的读时序图

图 32.2.5 就是镁光 DDR3L 的读时序图，我们依次来看一下图中这四部分都是什么内容：

①、tRCD，前面已经说过了。

②、AL。

③、CL。

④、RL 为读潜伏期，RL=AL+CL。

5、tRC 参数

tRC 是两个 ACTIVE 命令，或者 ACTIVE 命令到 REFRESH 命令之间的周期，DDR3L 数据手册会给出这个值，比如 NT5CC256M16EP-EK 的 tRC 值为 47.91ns，参考图 23.2.2。

6、tRAS 参数

tRAS 是 ACTIVE 命令到 PRECHARGE 命令之间的最小时间，DDR3L 的数据手册同样也会给出此参数，NT5CC256M16EP-EK 的 tRAS 值为 34ns，参考图 23.2.2。

23.3 I.MX6U MMDC 控制器简介

23.3.1 MMDC 控制器

学过 STM32 的同学应该记得，STM32 的 FMC 或 FSMC 外设用于连接 SRAM 或 SDRAM，对于 I.MX6U 来说也有 DDR 内存控制器，否则的话它怎么连接 DDR 呢？MMDC 就是 I.MX6U 的内存控制器，MMDC 是一个多模的 DDR 控制器，可以连接 16 位宽的 DDR3/DDR3L、16 位宽的 LPDDR2，MMDC 是一个可配置、高性能的 DDR 控制器。MMDC 外设包含一个内核(MMDC_CORE)和 PHY(MMDC_PHY)，内核和 PHY 的功能如下：

MMDC 内核：内核负责通过 AXI 接口与系统进行通信、DDR 命令生成、DDR 命令优化、读/写数据路径。

MMDC PHY：PHY 负责时序调整和校准，使用特殊的校准机制以保障数据能够在 400MHz 被准确捕获。

MMDC 的主要特性如下：

①、支持 DDR3/DDR3Lx16、支持 LPDDR2x16，不支持 LPDDR1MDDR 和 DDR2。

②、支持单片 256Mbit~8Gbit 容量的 DDR，列地址范围：8-12 位，行地址范围 11-16bit。2 个片选信号。

③、对于 DDR3，最大支持 8bit 的突发访问。

④、对于 LPDDR2 最大支持 4bit 的突发访问。

⑤、MMDC 最大频率为 400MHz，因此对应的数据速率为 800MT/S。

⑥、支持各种校准程序，可以自动或手动运行。支持 ZQ 校准外部 DDR 设备，ZQ 校准 DDR I/O 引脚、校准 DDR 驱动能力。

23.3.2 MMDC 控制器信号引脚

我们在使用 STM32 的时候 FMC/FSMC 的 IO 引脚是带有复用功能的，如果不接 SRAM 或 SDRAM 的话 FMC/FSMC 是可以用作其他外设 IO 的。但是，对于 DDR 接口就不一样了，因为 DDR 对于硬件要求非常严格，因此 DDR 的引脚都是独立的，一般没有复用功能，只做为 DDR 引脚使用。I.MX6U 也有专用的 DDR 引脚，如图 23.3.2.1 所示：

Signal	Description	Pad	Mode	Direction
DRAM_ADDR[15:0]	Address Bus Signals	DRAM_A[15:0]	No Muxing	O
DRAM_CAS	Column Address Strobe Signal	DRAM_CAS	No Muxing	O
DRAM_CS[1:0]	Chip Selects	DRAM_CS[1:0]	No Muxing	O
DRAM_DATA[31:0]	Data Bus Signals	DRAM_D[31:0]	No Muxing	I/O
DRAM_DQM[1:0]	Data Mask Signals	DRAM_DQM[1:0]	No Muxing	O
DRAM_ODT[1:0]	On-Die Termination Signals	DRAM_SDODT[1:0]	No Muxing	O
DRAM_RAS	Row Address Strobe Signal	DRAM_RAS	No Muxing	O
DRAM_RESET	Reset Signal	DRAM_RESET	No Muxing	O
DRAM_SDBA[2:0]	Bank Select Signals	DRAM_SDBA[2:0]	No Muxing	O
DRAM_SDCKE[1:0]	Clock Enable Signals	DRAM_SDCKE[1:0]	No Muxing	O
DRAM_SDCLK0_N	Negative Clock Signals	DRAM_SDCLK_[1:0]	No Muxing	O
DRAM_SDCLK0_P	Positive Clock Signals	DRAM_SDCLK_[1:0]	No Muxing	O
DRAM_SDQS[1:0]_N	Negative DQS Signals	DRAM_SDQS[1:0]_N	No Muxing	I/O
DRAM_SDQS[1:0]_P	Positive DQS Signals	DRAM_SDQS[1:0]_P	No Muxing	I/O
DRAM_SDWE	WE signal	DRAM_SDWE	No Muxing	O
DRAM_ZQPAD	ZQ signal	DRAM_ZQPAD	No Muxing	O

图 23.3.2.1 DDR 信号引脚

由于图 23.3.2.1 中的引脚是 DDR 专属的，因此就不存在缩为的 DDR 引脚复用配置，只需要设置 DDR 引脚的电气属性即可，注意，DDR 引脚的电气属性寄存器和普通的外设引脚电气属性寄存器不同！

23.3.3 MMDC 控制器时钟源

前面说了很多次，I.MX6U 的 DDR 或者 MDDC 的时钟频率为 400MHz，那么这 400MHz 时钟源怎么来的呢？这个就要查阅 I.MX6ULL 参考手册的《Chapter 18 Clock Controller Module(CCM)》章节。MMDC 时钟源如图 23.3.3.1 所示：

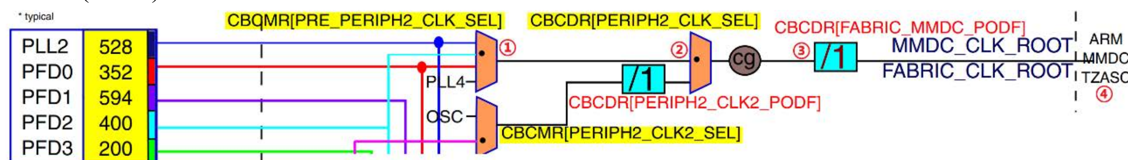


图 23.3.3.1 MMDC 时钟源

图 23.3.3.1 就是 MMDC 的时钟源路径图，主要分为 4 部分，我们依次来看一下每部分所组的工作：

①、pre_periph2 时钟选择器，也就是 periph2_clkd 的前级选择器，由 CBCMR 寄存器的 PRE_PERIPH2_CLK_SEL 位(bit22:21)来控制，一共有四种可选方案，如表 23.3.3.1 所示：

PRE_PERIPH2_CLK_SEL(bit22:21)	时钟源
00	PLL2
01	PLL2_PFD2
10	PLL2_PFD0
11	PLL4

表 23.3.3.1 pre_periph2 时钟源

从表 23.3.3.1 可以看出，当 PRE_PERIPH2_CLK_SEL 为 0x1 的时候选中 PLL2_PFD2 为 pre_periph2 时钟源。在前面的《第十六章 主频和时钟配置》中我们已经将 PLL2_PFD2 设置为 396MHz(约等于 400MHz)，I.MX6U 内部 boot rom 就是设置 PLL2_PFD2 作为 MMDC 的最终时钟源，这就是 I.MX6U 的 DDR 频率为 400MHz 的原因。

②、periph2_clk 时钟选择器，由 CBCDR 寄存器的 PERIPH2_CLK_SEL 位(bit26)来控制，当为 0 的时候选择 pll2_main_clk 作为 periph2_clk 的时钟源，当为 1 的时候选择 periph2_clk2_clk 作为 periph2_clk 的时钟源。这里肯定要将 PERIPH2_CLK_SEL 设置为 0，也就是选择 pll2_main_clk 作为 periph2_clk 的时钟源，因此 periph2_clk=PLL2_PFD0=396MHz。

③、最后就是分频器，由 CBCDR 寄存器的 FABRIC_MMDC_PODF 位(bit5:3)设置分频值，可设置 0~7，分别对应 1~8 分频，要配置 MMDC 的时钟源为 396MHz，那么此处就要设置为 1 分频，因此 FABRIC_MMDC_PODF=0。

以上就是 MMDC 的时钟源设置，I.MX6U 参考手册一直说 DDR 的频率为 400MHz，但是实际只有 396MHz，就和 NXP 宣传自己的 I.MX6ULL 有 800MHz 一样，实际只有 792MHz。

23.4 ALPHA 开发板 DDR3L 原理图

ALPHA 开发板有 EMMC 和 NAND 两种核心板，EMMC 核心板使用的 DDR3L 的型号为 NT5CC256M16EP-EK，容量为 512MB。NAND 核心板使用的 DDR3L 型号为 NT5CC128M16JR-EK，容量为 256MB，这两种型号的 DDR3L 封装一摸一样，有人可能就有疑问了，容量不同的

话地址线是不同的，比如行地址和列地址线数就不同，没错！但是 DDR3L 厂商为了方便选择将不同容量的 DDR3 封装做成一样，没有用到的地址线 DDR3L 芯片会屏蔽掉。而且，根据规定，所有厂商的 DDR 芯片 IO 一摸一样，不管是引脚定义还是引脚间距，但是芯片外形大小可能不同。因此只要做好硬件，可以在不需要修改硬件 PCB 的前提下，随意的更换不同容量、不同品牌的 DDR3L 芯片，极大的方便了我们的芯片选型。

正点原子 ALPHA 开发板 EMMC 和 NAND 核心板的 DDR3L 原理图一样，如图 23.4.1 所示：

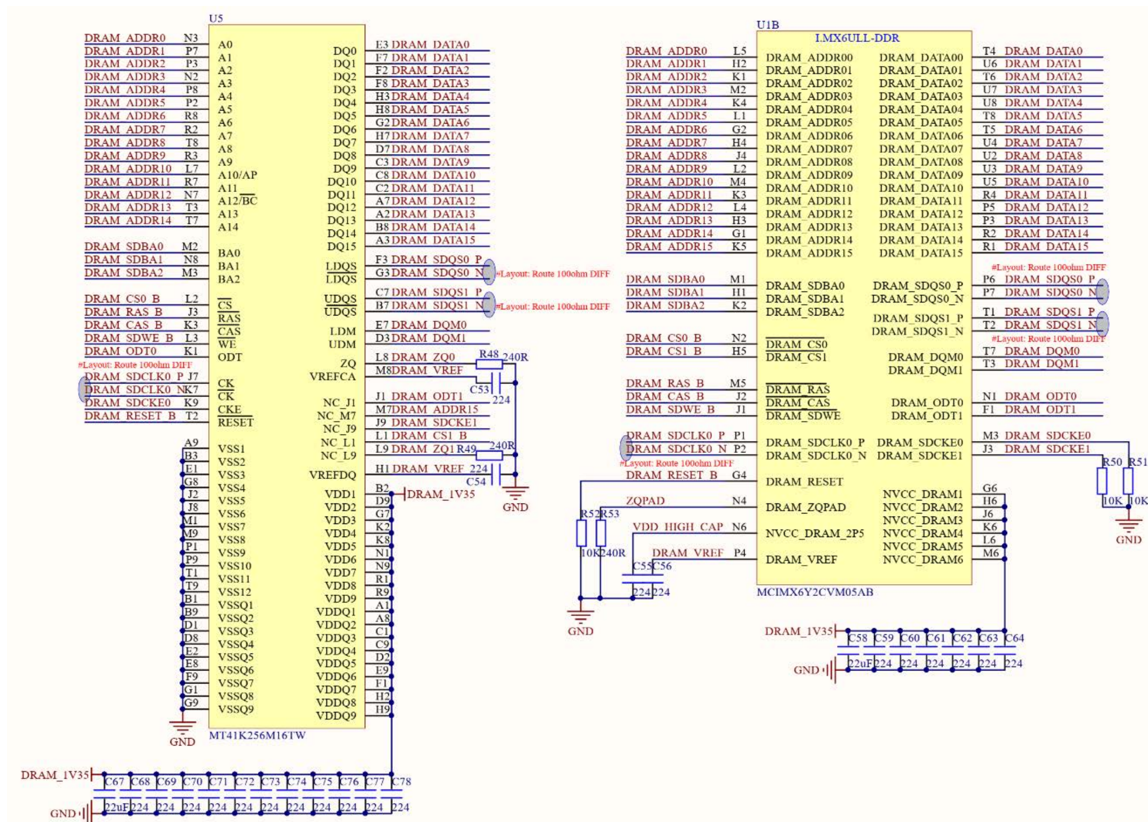


图 23.4.1 DDR3L 原理图

图 23.4.1 中左侧是 DDR3L 原理图，可以看出图中 DDR3L 的型号为 MT41K256M16TW，这个是镁光的 512MB DDR3L。但是我们实际使用的 512MB DDR3L 型号为 NT5CC256M16EP-EK，不排除以后可能会更换 DDR3L 型号，更换 DDR3L 芯片是不需要修改 PCB。图 23.4.1 中右边的是 I.MX6U 的 MMDC 控制器 IO。

23.5 DDR3L 初始化与测试

23.5.1 ddr_stress_tester 简介

NXP 提供了一个非常好用的 DDR 初始化工具，叫做 ddr_stress_tester。此工具已经放到了开发板光盘中，路径为：5、开发工具->6、NXP 官方 DDR 初始化与测试工具->ddr_stress_tester_v2.90_setup.exe.zip，我们简单介绍一下 ddr_stress_tester 工具，此工具特点如下：

①、此工具通过 USB OTG 接口与开发板相连接，也就是通过 USB OTG 口进行 DDR 的初始化与测试。

②、此工具有一个默认的配置文件，为 excel 表，通过此表可以设置板子的 DDR 信息，最

后生成一个.inc 结尾的 DDR 初始化脚本文件。这个.inc 文件就包含了 DDR 的初始化信息，一般都是寄存器地址和对应的寄存器值。

③、此工具会加载.inc 表里面的 DDR 初始化信息，然后通过 USB OTG 接口向板子下载 DDR 相关的测试代码，包括初始化代码。

④、对此工具进行简单的设置，即可开始 DDR 测试，一般要先做校准，因为不同的 PCB 其结构肯定不同，必须要做一次校准，校准完成以后会得到两个寄存器对应的校准值，我们需要用这个新的校准值来重新初始化 DDR。

⑤、此工具可以测试板子的 DDR 超频性能，一般认为 DDR 能够以超过标准工作频率 10%~20%稳定工作的话就认定此硬件 DDR 走线正常。

⑥、此工具也可以对 DDR 进行 12 小时的压力测试。

我们来看一下正点原子开发板光盘里面 5、开发工具->6、NXP 官方 DDR 初始化与测试工具目录下的文件，如图 23.5.1.1 所示：

ALIENTEK_256MB.inc	2019-06-06 20:16	INC 文件	8 KB
ALIENTEK_512MB.inc	2019-06-06 18:06	INC 文件	8 KB
ddr_stress_tester_v2.90_setup.exe.zip	2018-07-31 11:47	360压缩 ZIP 文件	2,207 KB
I.MX6UL_DDR3_Script_Aid_V0.02.xlsx	2018-07-31 12:08	Microsoft Excel 工...	84 KB
MX6X_DDR3_调校_应用手册_V4_20150730....	2018-08-11 9:38	Foxit Reader PDF D...	1,326 KB
飞思卡尔i.MX6平台DRAM接口高阶应用指导-...	2018-07-31 11:54	Foxit Reader PDF D...	3,164 KB

图 23.5.1.1 XP 官方 DDR 初始化与测试工具目录下的文件

我们依次来看一下图 23.5.1.1 中的这些文件的作用：

①、ALIENTEK_256MB.inc 和 ALIENTEK_512MB.inc，这两个就是通过 excel 表配置生成的，针对正点原子开发板的 DDR 配置脚本文件。

②、ddr_stress_tester_v2.90_setup.exe.zip 就是我们要用的 ddr_stress_tester 软件，大家自行安装即可，一定要记得安装路径。

③、I.MX6UL_DDR3_Script_Aid_V0.02.xlsx 就是 NXP 编写的针对 I.MX6UL 的 DDR 初始化 execl 文件，可以在此文件里面填写 DDR 的相关参数，然后就会生成对应的.inc 初始化脚本。

④、最后两个 PDF 文档就是关于 I.MX6 系列的 DDR 调试文档，这两个是 NXP 编写的。

23.5.2 DDR3L 驱动配置

1、安装 ddr_stress_tester

首先要安装 ddr_stress_testr 软件，安装方法很简单，这里就不做详细的讲解了。但是一定要记得安装路径！因为我们要到安装路径里面找到测试软件。比如我安装到了 D:\Program Files (x86)里面，安装完成以后就会在此目录下生成一个名为 ddr_stress_tester_v2.90 的文件夹，此文件夹就是 DDR 测试软件，进入到此文件夹中，里面的文件如图 23.5.2.1 所示：

bin	2019-06-06 18:39	文件夹	
log	2019-06-06 18:39	文件夹	
script	2019-06-06 18:39	文件夹	
DDR_Tester.exe	2017-08-02 10:44	应用程序	3,451 KB
LA_OPT_Base_License.html	2018-07-06 13:37	360 se HTML Docu...	195 KB
SCR-ddr_stress_tester_v2.9.0.txt	2018-07-06 15:10	文本文档	4 KB

图 23.5.2.1 ddr_stress_tester 安装文件

图 23.5.2.1 中的 DDR_Tester.exe 就是我们稍后要使用的 DDR 测试软件。

2、配置 DDR3L，生成初始化脚本

将开发板光盘中的：5、开发工具->6、NXP 官方 DDR 初始化与测试工具->I.MX6UL_DDR3_Script_Aid_V0.02.xlsx 文件拷贝到 ddr_stress_testr 软件安装目录中，完成以后如图 23.5.2.2 所示：

名称	修改日期	类型	大小
bin	2019-06-06 18:39	文件夹	
log	2019-06-06 18:39	文件夹	
script	2019-06-06 18:39	文件夹	
DDR_Tester.exe	2017-08-02 10:44	应用程序	3,451 KB
LA_OPT_Base_License.html	2018-07-06 13:37	360 se HTML Docu...	195 KB
SCR-ddr_stress_tester_v2.9.0.txt	2018-07-06 15:10	文本文档	4 KB
I.MX6UL_DDR3_Script_Aid_V0.02.xlsx	2018-07-31 12:08	Microsoft Excel 工...	84 KB

I.MX6U的DDR3配置excel表

图 23.5.2.2 拷贝完成以后的测试软件目录

I.MX6UL_DDR3_Script_Aid_V0.02.xlsx 就是 NXP 为 I.MX6UL 编写的 DDR3 配置 excel 表，虽然看名字是为 I.MX6UL 编写的，但是 I.MX6ULL 也是可以使用的。

打开 I.MX6UL_DDR3_Script_Aid_V0.02.xlsx，打开以后如图 23.5.2.3 所示：

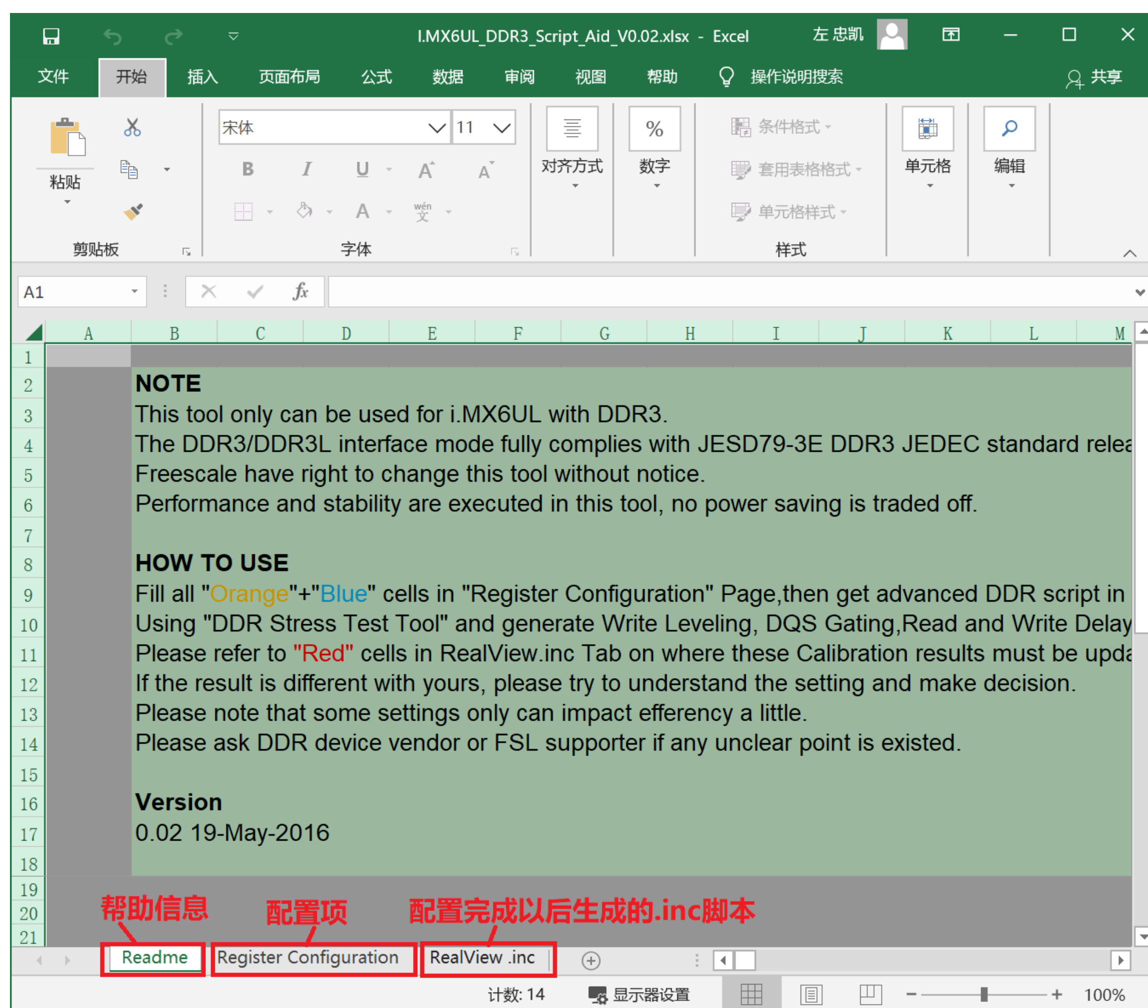


图 23.5.2.3 配置 excel 表

图 23.5.2.3 中最下方有三个选项卡，这三个选项卡的功能如下：

①、Readme 选项卡，此选项卡是帮助信息，告诉用户此文件如何使用。

②、Register Configuration 选项卡，顾名思义，此选项卡用于完成寄存器配置，也就是配置

DDR3，此选项卡是我们重点要讲解的。

③、RealView.inc 选项卡，当我们配置好 Register Configuration 选项卡以后，RealView.inc 选项卡里面就保存着寄存器地址和对应的寄存器值。我们需要另外新建一个后缀为.inc 的文件来保存 RealView.inc 中的初始化脚本内容，ddr_stress_testr 软件就是要使用此.inc 结尾的初始化脚本文件来初始化 DDR3。

选中“Register Configuration”选项卡，如图 23.5.2.4 所示：

Device Information	
Manufacturer:	Micron
Memory part number:	MT41K256M16HA-125
Memory type:	DDR3-1600
DRAM density (Gb)	4
DRAM Bus Width	16
Number of Banks	8
Number of ROW Addresses	15
Number of COLUMN Addresses	10
Page Size (K)	2
Self-Refresh Temperature (SRT)	Extended
tRCD=tRP=CL (ns)	13.75
tRC Min (ns)	48.75
tRAS Min (ns)	35
System Information	
i.Mx Part	i.MX6UL
Bus Width	16
Density per chip select (Gb)	4
Number of Chip Selects used	1
Total DRAM Density (Gb)	4
DRAM Clock Freq (MHz)	400
DRAM Clock Cycle Time (ns)	2.5
Address Mirror (for CS1)	Disable
SI Configuration	
DRAM DSE Setting - DQ/DQM (ohm)	48
DRAM DSE Setting - ADDR/CMD/CTL (ohm)	48
DRAM DSE Setting - CK (ohm)	48
DRAM DSE Setting - DQS (ohm)	48
System ODT Setting (ohm)	60

Readme　Register Configuration　RealView .inc　⊕

图 23.5.2.4 配置界面

图 23.5.2.4 就是具体的配置界面，主要分为三部分：

①、Device Information

DDR3 芯片设备信息设置，此部分需要根据所使用的 DDR3 芯片来设置，具体的设置项如下：

Manufacturer：DDR3 芯片厂商，默认为镁光(Micron)，这个没有意义，比如我们用的 nanya 的 DDR3，但是此配置文件也是可以使用的。

Memory part number：DDR3 芯片型号，可以不用设置，没有实际意义。

Memory type:DDR3 类型，有 DDR3-800、DDR3-1066、DDR3-1333 和 DDR3-1600，在此选项右侧有个下拉箭头，点击下拉箭头即可查看所有的可选选项，如图 23.5.2.5 所示：

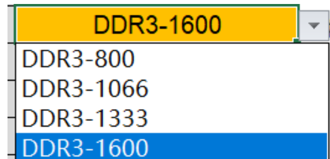


图 23.5.2.5 Memory type 可选选项

从图 23.5.2.5 可以看出，最大只能选择 DDR3-1600，没有 DDR3-1866 选项，因此我们就只能选择 DDR3-1600。

DRAM density(Gb)：DDR3 容量，根据实际情况选择，同样右边有个下拉箭头，打开下拉箭头即可看到所有可选的容量，如图 23.5.2.6 所示：

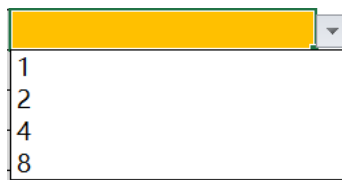


图 23.5.2.6 容量选择

从图 23.5.2.6 可以看出，可选的容量为 1、2、4 和 8Gb，如果使用的 512MB 的 DDR3 就应该选择 4，如果使用的 256MB 的 DDR3 就应该选择 2。

DRAM Bus width：DDR3 位宽，可选的选项如图 23.5.2.7 所示：

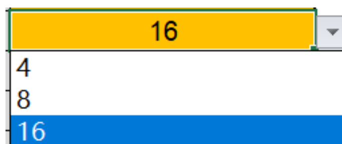


图 23.5.2.7 DDR3 位宽

正点原子 ALPHA 开发板所有的 DDR3 都是 16 位宽，因此选择 16。

Number of Banks：DDR3 内部 BANK 数量，对于 DDR3 来说内部都是 8 个 BANK，因此固定为 8。

Number of ROW Addresses：行地址宽度，可选 11~16 位，这个要具体所使用的 DDR3 芯片来定，如果是 EMMC 核心板(DDR3 型号为 NT5CC256M16EP-EK)，那么行地址为 15 位。如果是 NAND 核心板(DDR3 型号为 NT5CC128M16JR-EK)，行地址就为 14 位。

Number COLUMN Addresses：列地址宽度，可选 9~12 位。如果是 EMMC 核心板(DDR3 型号为 NT5CC256M16EP-EK)，那么列地址为 10 位。如果是 NAND 核心板(DDR3 型号为 NT5CC128M16JR-EK)，行地址就为 10 位。

Page Size(K)：DDR3 页大小，可选 1 和 2，NT5CC256M16EP-EK 和 NT5CC128M16JR-EK 的页大小都为 2KB，因此选择 2。

Self-Refresh Temperature(SRT)：固定为 Extended，不需要修改。

tRCD=tRP=CL(ns)：DDR3 的 tRCD-tRP-CL 时间参数，要查阅所使用的 DDR3 芯片手册，NT5CC256M16EP-EK 和 NT5CC128M16JR-EK 都为 13.91ns，因此在后面填写 13.91。

tRC Min(ns)：DDR3 的 tRC 时间参数，NT5CC256M16EP-EK 和 NT5CC128M16JR-EK 都为 47.91ns，因此在后面填写 47.91。

tRAS Min(ns)：DDR3 的 tRAS 时间参数，NT5CC256M16EP-EK 和 NT5CC128M16JR-EK 都为 34ns，因此在后面填写 34。

②、System Information

此部分设置 I.MX6UL/6ULL 相关属性，具体的设置项如下：

i.Mx Part： 固定为 i.MX6UL。

Bus Width： 总线宽度，16 位宽。

Density per Chip select(Gb)： 每个片选对应的 DDR3 容量，可选 1~16，根据实际所使用的 DDR3 芯片来填写，512MB 的话就选择 4，256MB 的话就选择 2。

Number of Chip Select used： 使用几个片选信号？可选择 1 或 2，正点原子所有的核心板都只使用了一个片选信号，因此选择 1。

Total DRAM Density(Gb)： 整个 DDR3 的容量，单位为 Gb，如果是 512MB 的话就是 4，如果是 256MB 的话就是 2。

DRAM Clock Freq(MHz)： DDR3 工作频率，设置为 400MHz。

DRAM Clock Cycle Time(ns)： DDR3 工作频率对应的周期，单位为 ns，如果工作在 400MHz，那么周期就是 2.5ns。

Address Mirror(for CS1)： 地址镜像，仅 CS1 有效，此处选择关闭，也就是"Disable"，此选项我们不需要修改。

③、SI Configuratin

此部分是信号完整性方面的配置，主要是一些信号线的阻抗设置，这个要咨询硬件工程师，这里我们直接使用 NXP 的默认设置即可。

关于 DDR3 的配置我们就讲解到这里，如果是 EMMC 核心板(DDR3 型号为 NT5CC256M16EP-EK)，那么配置如图 23.5.2.8 所示：

Device Information	
Manufacturer:	Micron
Memory part number:	MT41K256M16HA-125
Memory type:	DDR3-1600
DRAM density (Gb)	4
DRAM Bus Width	16
Number of Banks	8
Number of ROW Addresses	15
Number of COLUMN Addresses	10
Page Size (K)	2
Self-Refresh Temperature (SRT)	Extended
tRCD=tRP=CL (ns)	13.91
tRC Min (ns)	47.91
tRAS Min (ns)	34
System Information	
i.Mx Part	i.MX6UL
Bus Width	16
Density per chip select (Gb)	4
Number of Chip Selects used	1
Total DRAM Density (Gb)	4
DRAM Clock Freq (MHz)	400
DRAM Clock Cycle Time (ns)	2.5
Address Mirror (for CS1)	Disable
SI Configuration	
DRAM DSE Setting - DQ/DQM (ohm)	48
DRAM DSE Setting - ADDR/CMD/CTL (ohm)	48
DRAM DSE Setting - CK (ohm)	48
DRAM DSE Setting - DQS (ohm)	48
System ODT Setting (ohm)	60

Readme | Register Configuration | RealView .inc

图 23.5.2.8 EMMC 核心板配置。

NAND 核心板配置(DDR3 型号为 NT5CC128M16JR-EK)配置如图 23.5.2.9 所示：

Device Information	
Manufacturer:	Micron
Memory part number:	MT41K256M16HA-125
Memory type:	DDR3-1600
DRAM density (Gb)	2
DRAM Bus Width	16
Number of Banks	8
Number of ROW Addresses	14
Number of COLUMN Addresses	10
Page Size (K)	2
Self-Refresh Temperature (SRT)	Extended
tRCD=tRP=CL (ns)	13.91
tRC Min (ns)	47.91
tRAS Min (ns)	34
System Information	
i.Mx Part	i.MX6UL
Bus Width	16
Density per chip select (Gb)	2
Number of Chip Selects used	1
Total DRAM Density (Gb)	2
DRAM Clock Freq (MHz)	400
DRAM Clock Cycle Time (ns)	2.5
Address Mirror (for CS1)	Disable
SI Configuration	
DRAM DSE Setting - DQ/DQM (ohm)	48
DRAM DSE Setting - ADDR/CMD/CTL (ohm)	48
DRAM DSE Setting - CK (ohm)	48
DRAM DSE Setting - DQS (ohm)	48
System ODT Setting (ohm)	60

Readme | Register Configuration | RealView .inc

图 23.5.2.9 NAND 核心板配置

后面我就以 EMMC 核心板为例讲解了，配置完成以后点击 RealView.inc 选项卡，如图 23.5.2.10 所示：

```
//=============================================================================
//init script for i.MX6UL DDR3
//=============================================================================
// Revision History
// v01
//=============================================================================

wait = on
//=============================================================================
// Disable              WDOG
//=============================================================================
setmem /16              0x020bc000 =    0x30

//=============================================================================
// Enable all clocks (they are disabled by ROM code)
//=============================================================================
setmem /32              0x020c4068 =    0xffffffff
setmem /32              0x020c406c =    0xffffffff
setmem /32              0x020c4070 =    0xffffffff
setmem /32              0x020c4074 =    0xffffffff
setmem /32              0x020c4078 =    0xffffffff
setmem /32              0x020c407c =    0xffffffff
setmem /32              0x020c4080 =    0xffffffff


//=============================================================================
// IOMUX
//=============================================================================
```

Readme | Register Configuration | RealView .inc

图 23.5.2.10 生成的配置脚本。

图 23.5.2.10 中的 RealView.inc 就是生成的配置脚本，全部是“寄存器地址=寄存器值”这

种形式。RealView.inc 不能直接用，我们需要新建一个以.inc 结尾的文件，名字自定义，比如我名为“ALIENTEK_512MB”的.inc 文件，如图 23.5.2.11 所示：

bin	2019-06-06 18:39	文件夹	
log	2019-06-06 18:39	文件夹	
script	2019-06-06 18:39	文件夹	
DDR_Tester.exe	2017-08-02 10:44	应用程序	3,451 KB
LA_OPT_Base_License.html	2018-07-06 13:37	360 se HTML Docu...	195 KB
SCR-ddr_stress_tester_v2.9.0.txt	2018-07-06 15:10	文本文档	4 KB
I.MX6UL_DDR3_Script_Aid_V0.02.xlsx	2018-07-31 12:08	Microsoft Excel 工...	84 KB
ALIENTEK_512MB.inc	2019-10-06 17:17	INC 文件	0 KB

新建的.inc文件

图 23.5.2.11 新建.inc 文件

用 notepad++打开 ALIENTEK_512MB.inc 文件，然后将图 23.5.2.10 中 RealView.inc 里面的所有内容全部拷贝到 ALIENTEK_512MB.inc 文件中，完成以后如图 23.5.2.12 所示：

D:\Program Files (x86)\ddr_stress_tester_v2.90\ALIENTEK_512MB.inc - Notepad++

```
//==========================================================================
//init script for i.MX6UL DDR3
//==========================================================================
// Revision History
// v01
//==========================================================================

wait = on
//==========================================================================
// Disable  WDOG
//==========================================================================
setmem /16  0x020bc000 =    0x30

//==========================================================================
// Enable all clocks (they are disabled by ROM code)
//==========================================================================
setmem /32  0x020c4068 =    0xffffffff
setmem /32  0x020c406c =    0xffffffff
setmem /32  0x020c4070 =    0xffffffff
setmem /32  0x020c4074 =    0xffffffff
```

Pascal source file　length : 7,557　lines : 208　Ln : 6　Col : 30　Sel : 0 | 0　Windows (CR LF)　UTF-8　IN

图 23.5.2.12 完成后的 ALIENTEK_512MB.inc 文件内容

至此，DDR3 配置就全部完成，DDR3 的配置文件 ALIENTEK_512MB.inc 已经得到了，接下来就是使用此配置文件对正点原子 ALPHA 开发板的 DDR3 进行校准并进行超频测试。

23.5.3 DDR3L 校准

首先要用 DDR_Tester.exe 软件对正点原子 ALPAH 开发板的 DDR3L 进行校准，因为不同的 PCB 其走线不同，必须要进行校准，经过校准一会 DDR3L 就会工作到最佳状态。

1、将开发板通过 USB OTG 线连接到电脑上

DDR_Tester 软件通过 USB OTG 线将测试程序下载到开发板中，因此首先需要使用 USB OTG 线将开发板和电脑连接起来，如图 23.5.3.1 所示：

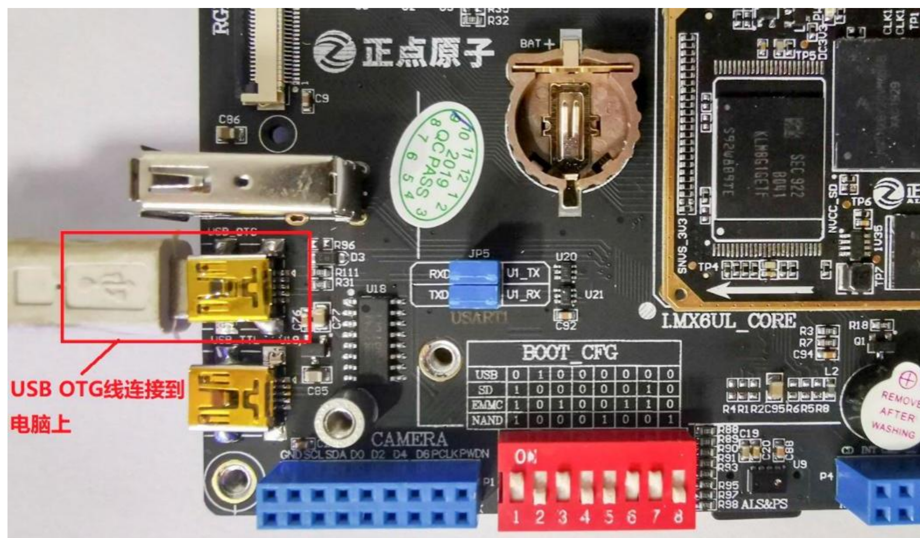


图 23.5.3.1 USB OTG 连接示意图

USB OTG 线连接成功以后还需要如下两步：

①、弹出 TF 卡，如果插入了 TF 卡，那么一定要弹出来!!

②、设置拨码开关从 USB 启动，如图 23.5.3.2 所示：

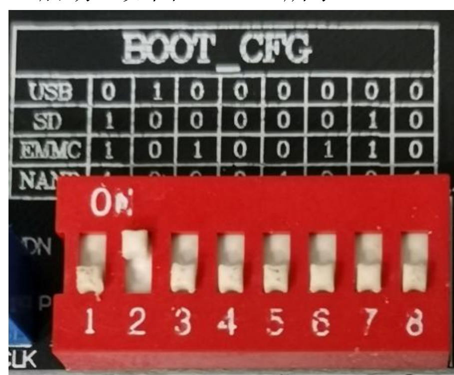


图 23.5.3.2 USB 启动

2、DDR_Tester 软件

双击“DDR_Tester.exe”，打开测试软件，如图 23.5.3.3 所示：

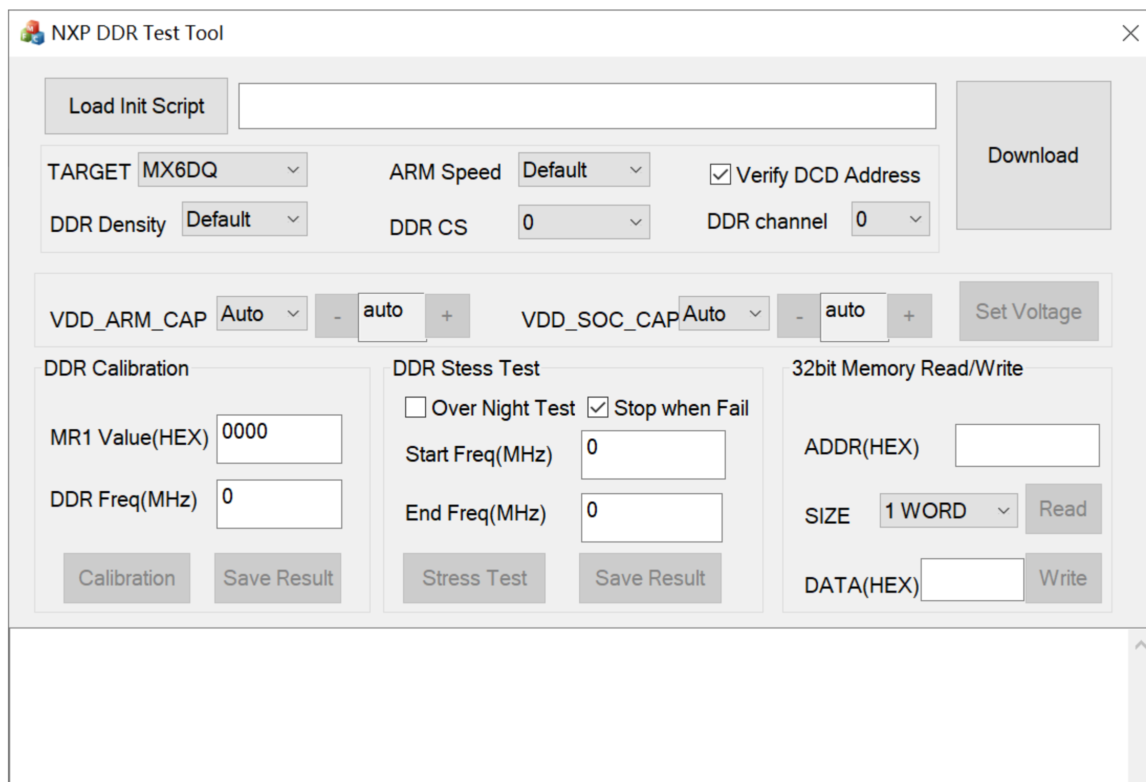


图 23.5.3.3 NXP DDR Test Tool

点击图 23.5.3.3 中的“Load init Script”加载前面已经生成的初始化脚本文件 ALIENTEK_512MB.inc，完成以后如图 23.5.3.4 所示：

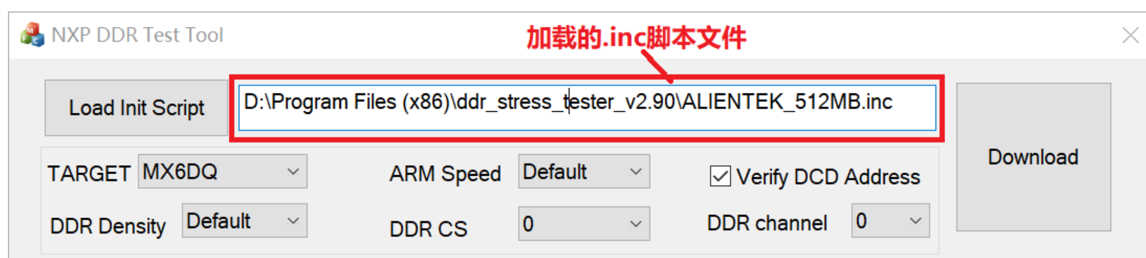


图 23.5.3.4 .inc 文件加载成功后的界面

ALIENTEK_512MB.inc 文件加载成功以后还不能直接用，还需要对 DDR Test Tool 软件进行设置，设置完成以后如图 23.5.3.5 所示：

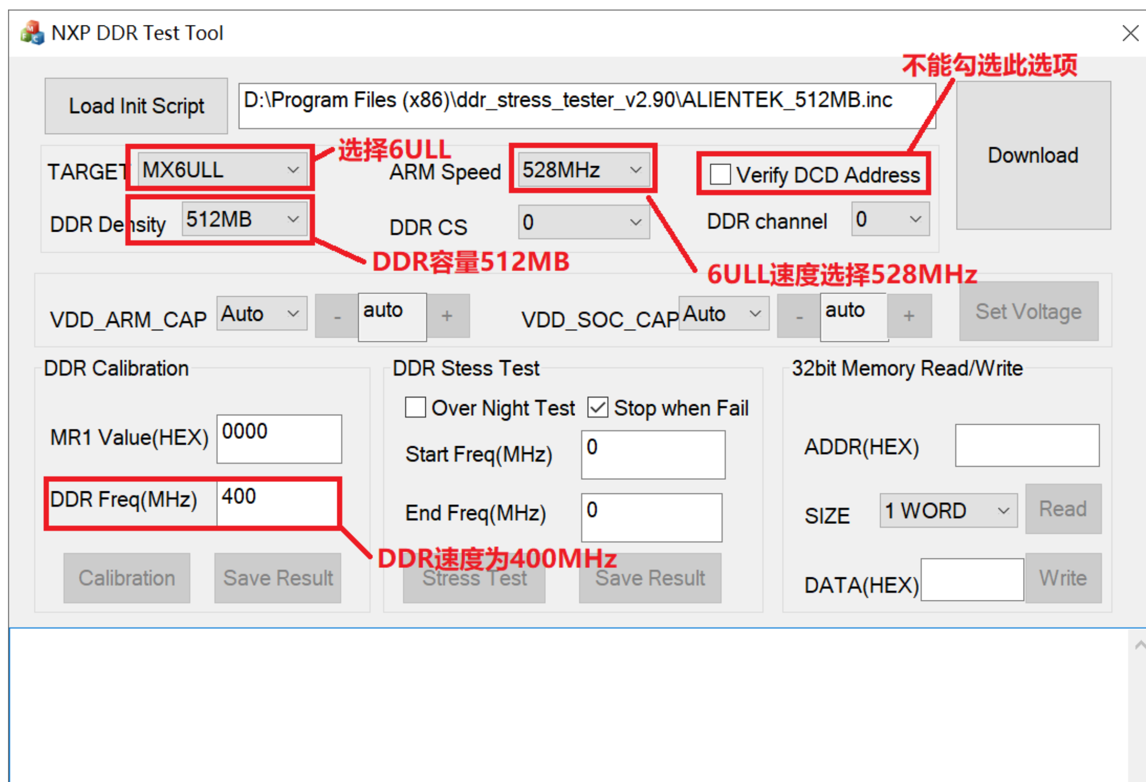


图 23.5.3.5 DDR Test Tool 配置

一切设置好以后点击图 23.5.3.5 中右上方大大的“Download”按钮，将测试代码下载到开发板中(具体下载到哪里笔者也不清楚，估计是 I.MX6ULL 内部的 OCRAM)，下载完成以后 DDR Test Tool 下方的信息窗口就会输出一些内容，如图 23.5.3.6 所示：

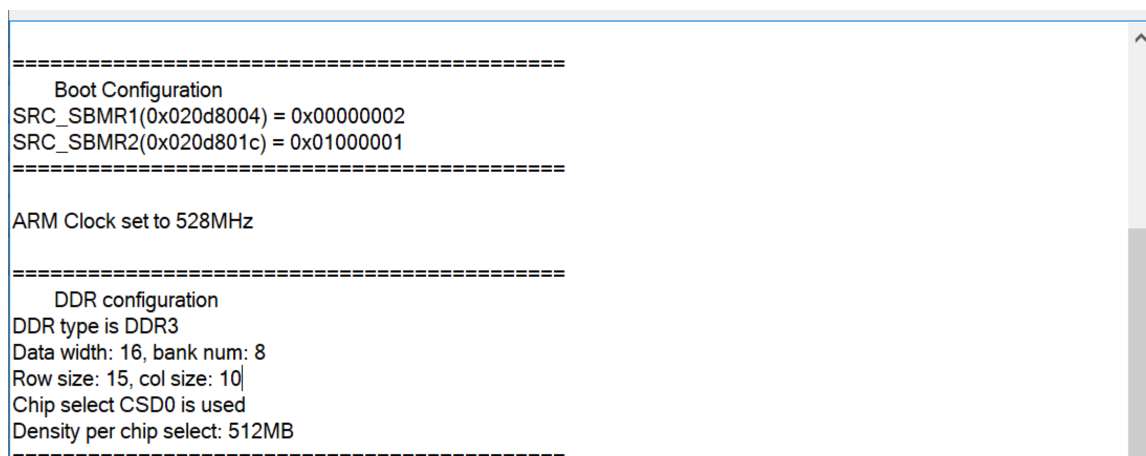


图 23.5.3.6 信息输出

图 23.5.3.6 输出了一些关于板子的信息，比如 SOC 型号、工作频率、DDR 配置信息等等。DDR Test Tool 工具有三个测试项：DDR Calibration、DDR Stess Test 和 32bit Memory Read/Write，我们首先要做校准测试，因为不同的 PCB、不同的 DDR3L 芯片对信号的影响不同，必须要进行校准，然后用新的校准值重新初始化 DDR。点击“Calibraton”按钮，如图 23.5.3.7 所示：

图 23.5.3.7 开始校准

点击图 23.5.3.7 中的“Calibration”按钮以后就会自动开始校准，最终会得到 Write leveling calibtarion、Read DQS Gating Calibration、Read calibration 和 Write calibration，一共四种校准结果，校准结果如下：

示例代码 23.5.3.1 DDR3L 校准结果

```
1  Write leveling calibration
2  MMDC_MPWLDECTRL0 ch0 (0x021b080c) = 0x00000000
3  MMDC_MPWLDECTRL1 ch0 (0x021b0810) = 0x000B000B
4
5  Read DQS Gating calibration
6  MPDGCTRL0 PHY0 (0x021b083c) = 0x0138013C
7  MPDGCTRL1 PHY0 (0x021b0840) = 0x00000000
8
9  Read calibration
10 MPRDDLCTL PHY0 (0x021b0848) = 0x40402E34
11
12 Write calibration
13 MPWRDLCTL PHY0 (0x021b0850) = 0x40403A34
```

所谓的校准结果其实就是得到了一些寄存器对应的值，比如 MMDC_MPWLDECTRL0 寄存器地址为 0X021B080C，此寄存器是 PHY 写平衡延时寄存器 0，经过校准以后此寄存器的值应该为 0X00000000，以此类推。我们需要修改 ALIENTEK_512MB.inc 文件，找到 MMDC_MPWLDECTRL0、MMDC_MPWLDECTRL1、MPDGCTRL0 PHY0、MPDGCTRL1 PHY0、MPRDDLCTL PHY0 和 MPWRDLCTL PHY0 这 6 个寄存器，然后将其值改为示例代码 23.5.3.1 中的校准后的值。注意，在 ALIENTEK_512MB.inc 中可能找不到 MMDC_MPWLDECTRL1(0x021b0810)和 MPDGCTRL1 PHY0(0x021b0840)这两个寄存器，找不到就不用修改了。

ALIENTEK_512MB.inc 修改完成以后重新加载并下载到开发板中，至此 DDR 校准完成，校准的目的就是得到示例代码 23.5.3.1 中这 6 个寄存器的值！

23.5.4 DDR3L 超频测试

校准完成以后就可以进行 DDR3 超频测试，超频测试的目的就是为了检验 DDR3 硬件设计合不合理，一般 DDR3 能够超频到比标准频率高 10%~15%的话就认为硬件没有问题，因此对于正点原子的 ALPHA 开发板而言，如果 DDR3 能够超频到 440MHz~460MHz 那么就认为 DDR3 硬件工作良好。

DDR Test Tool 支持 DDR3 超频测试，只要指定起始频率和终止频率，那么工具就会自动开始一点点的增加频率，直到达到终止频率或者测试失败。设置如图 23.5.4.1 所示：

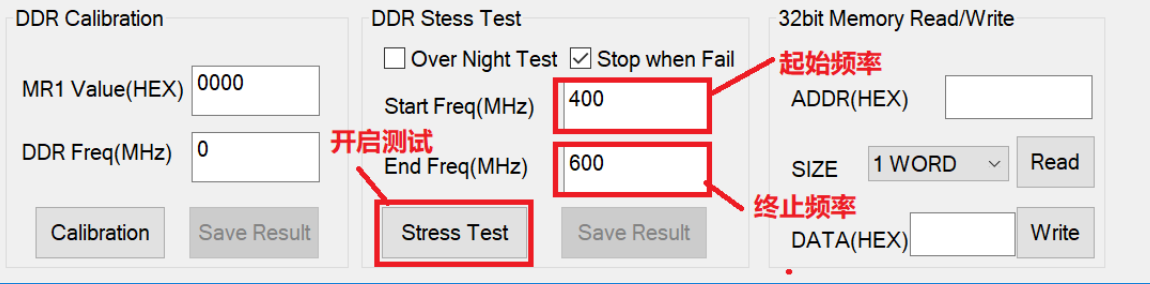


图 23.5.4.1 超频测试配置

图 23.5.4.1 中设置好其实频率为 400MHz，终止频率为 600MHz，设置好以后点击“Stress Test”开启超频测试，超频测试时间比较久，大家耐心等待测试结果即可。超频测试完成以后结果如图 23.5.4.2 所示(因为硬件不同，测试结果可能有些许区别)：

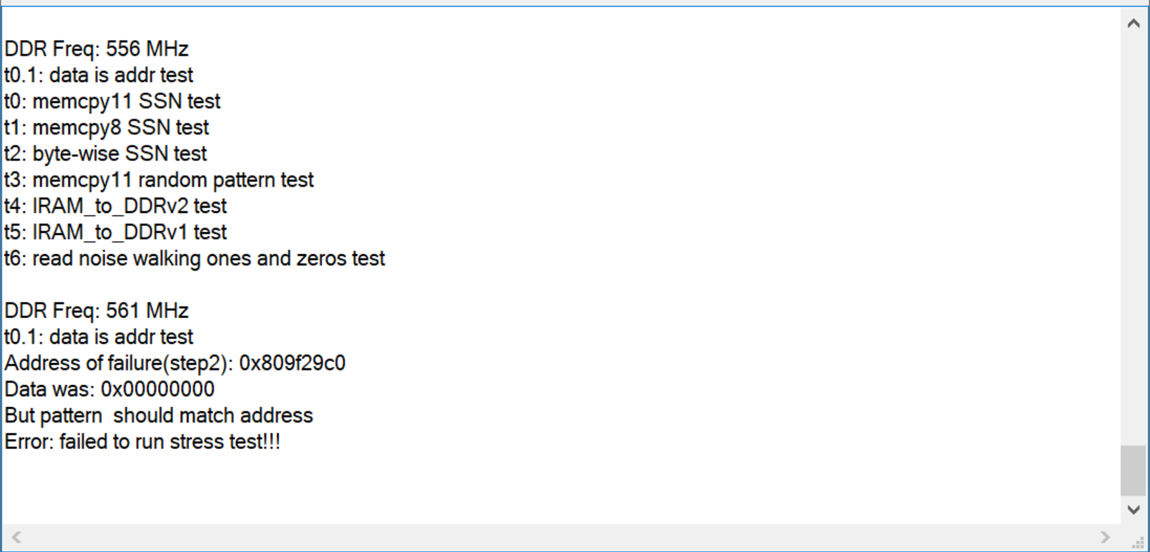


图 23.5.4.2 超频测试结果

从图 23.5.4.2 可以看出，正点原子的 ALPAH 开发板 EMMC 核心板 DDR3 最高可以超频到 556MHz，当超频到 561MHz 的时候就失败了。556MHz 超过了 460MHz，说明正点原子的 ALPHA 开发板 DDR3 硬件是没有任何问题的。

23.5.5 DDR3L 驱动总结

ALIENTEK_512MB.inc 就是我们最终得到的 DDR3L 初始化脚本，其中包括了时钟、IO 等初始化。I.MX6U 的 DDR3 接口关于 IO 有一些特殊的寄存器需要初始化，如表 23.5.5.1 所示：

寄存器地址	寄存器名	寄存器值
0X020E04B4	IOMUXC_SW_PAD_CTL_GRP_DDR_TYPE	0X000C0000
0X020E04AC	IOMUXC_SW_PAD_CTL_GRP_DDRPKE	0X00000000
0X020E027C	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDCLK_0	0X00000028
0X020E0250	IOMUXC_SW_PAD_CTL_PAD_DRAM_CAS	0X00000028
0X020E024C	IOMUXC_SW_PAD_CTL_PAD_DRAM_RAS	0X00000028
0X020E0490	IOMUXC_SW_PAD_CTL_GRP_ADDDS	0X00000028

0X020E0288	IOMUXC_SW_PAD_CTL_PAD_DRAM_RESET	0X00000028
0X020E0270	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDBA2	0X00000000
0X020E0260	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDODT0	0X00000028
0X020E0264	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDODT1	0X00000028
0X020E04A0	IOMUXC_SW_PAD_CTL_GRP_CTLDS	0X00000028
0X020E0494	IOMUXC_SW_PAD_CTL_GRP_DDRMODE_CTL	0X00020000
0X020e0280	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDQS0	0X00000028
0X020E0284	IOMUXC_SW_PAD_CTL_PAD_DRAM_SDQS1	0X00000028
0X020E04B0	IOMUXC_SW_PAD_CTL_GRP_DDRMODE	0X00020000
0X020e0498	IOMUXC_SW_PAD_CTL_GRP_B0DS	0X00000028
0X020E04A4	IOMUXC_SW_PAD_CTL_GRP_B1DS	0X00000028
0X020E0244	IOMUXC_SW_PAD_CTL_PAD_DRAM_DQM0	0X00000028
0X020E0248	IOMUXC_SW_PAD_CTL_PAD_DRAM_DQM1	0X00000028

表 23.5.5.1 DDR3 IO 相关初始化

接下来看一下 MMDC 外设寄存器初始化，如表 23.5.5.2 所示：

寄存器地址	寄存器名	寄存器值
0X021B0800	DDR_PHY_P0_MPZQHWCTRL	0XA1390003
0X021B080C	MMDC_MPWLDECTRL0	0X00000000
0X021B083C	MPDGCTRL0	0X0138013C
0X021B0848	MPRDDLCTL	0X40402E34
0X021B0850	MPWRDLCTL	0X40403A34
0X021B081C	MMDC_MPRDDQBY0DL	0X33333333
0X021B0820	MMDC_MPRDDQBY1DL	0X33333333
0X021B082C	MMDC_MPWRDQBY0DL	0XF3333333
0X021B0830	MMDC_MPWRDQBY1DL	0XF3333333
0X021B08C0	MMDC_MPDCCR	0X00921012
0X021B08B8	DDR_PHY_P0_MPMUR0	0X00000800
0X021B0004	MMDC0_MDPDC	0X0002002D
0X021B0008	MMDC0_MDOTC	0X1B333030
0X021B000C	MMDC0_MDCFG0	0X676B52F3
0X021B0010	MMDC0_MDCFG1	0XB66D0B63
0X021B0014	MMDC0_MDCFG2	0X01FF00DB
0X021b002c	MMDC0_MDRWD	0X000026D2
0X021b0030	MMDC0_MDOR	0X006B1023
0X021b0040	MMDC_MDASP	0X0000004F
0X021b0000	MMDC0_MDCTL	0X84180000
0X021b0890	MPPDCMPR2	0X00400a38
0X021b0020	MMDC0_MDREF	0X00007800
0X021b0818	DDR_PHY_P0_MPODTCTRL	0X00000227
0X021b0004	MMDC0_MDPDC	0X0002556D
0X021b0404	MMDC0_MAPSR	0X00011006

表 23.5.5.2 MMDC 外设寄存器初始化及初始化序列

关于 I.MX6U 的 DDR3 就讲解到这里，因为牵扯到的寄存器太多了，因此没有详细的去分析这些寄存器，大家感兴趣的可以对照着参考手册去分析各个寄存器的含义以及配置值。

第二十四章 RGBLCD 显示实验

LCD 液晶屏是常用到的外设，通过 LCD 可以显示绚丽的图形、界面等，提高人机交互的效率。I.MX6U 提供了一个 eLCDIF 接口用于连接 RGB 接口的液晶屏。本章我们就学习如何驱动 RGB 接口液晶屏，并且在屏幕上显示字符。

24.1 LCD 和 eLCDIF 简介

24.1.1 LCD 简介

LCD 全称是 Liquid Crystal Display，也就是液晶显示器，是现在最常用到的显示器，手机、电脑、各种人机交互设备等基本都用到了 LCD，最常见就是手机和电脑显示器了。由于笔者不是 LCD 从业人员，对于 LCD 的具体原理不了解，百度百科对于 LCD 的原理解释如下：

LCD 的构造是在两片平行的玻璃基板当中放置液晶盒，下基板玻璃上设置 TFT（薄膜晶体管），上基板玻璃上设置彩色滤光片，通过 TFT 上的信号与电压改变来控制液晶分子的转动方向，从而达到控制每个像素点偏振光出射与否而达到显示目的。

我们现在要在 I.MX6U-ALPHA 开发板上使用 LCD，所以不需要去研究 LCD 的具体实现原理，我们只需要从使用的角度去关注 LCD 的几个重要点：

1、分辨率

提起 LCD 显示器，我们都会听到 720P、1080P、2K 或 4K 这样的字眼，这个就是 LCD 显示器分辨率。LCD 显示器都是由一个一个的像素点组成，像素点就类似一个灯(在 OLED 显示器中，像素点就是一个小灯)，这个小灯是 RGB 灯，也就是由 R(红色)、G(绿色)和 B(蓝色)这三种颜色组成的，而 RGB 就是光的三原色。1080P 的意思就是一个 LCD 屏幕上的像素数量是 1920*1080 个，也就是这个屏幕一列 1080 个像素点，一共 1920 列，如图 24.1.1.1 所示：

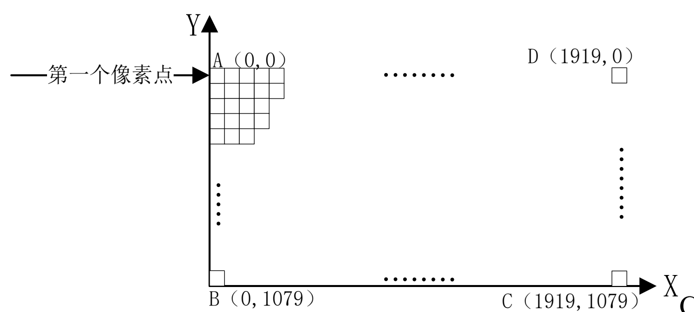


图 24.1.1.1 LCD 像素点排布

在图 24.1.1.1 就是 1080P 显示器的像素示意图，X 轴就是 LCD 显示器的横轴，Y 轴就是显示器的竖轴。图中的小方块就是像素点，一共有 1920*1080=2073600 个像素点。左上角的 A 点是第一个像素点，右下角的 C 点就是最后一个像素点。2K 就是 2560*1440 个像素点，4K 是 3840*2160 个像素点。很明显，在 LCD 尺寸不变的情况下，分辨率也高越清晰。同样的，分辨率不变的情况下，LCD 尺寸越小越清晰。比如我们常用的 24 寸显示器基本都是 1080P 的，而我们现在使用的 5 寸的手机基本也是 1080P 的，但是手机显示细腻程度就要比 24 寸的显示器要好很多！

由此可见，LCD 显示器的分辨率是一个很重要的参数，但是并不是分辨率越高的 LCD 就越好。衡量一款 LCD 的好坏，分辨率只是其中的一个参数，还有色彩还原程度、色彩偏离、亮度、可视角度、屏幕刷新率等其他参数。

2、像素格式

上面讲了，一个像素点就相当于一个 RGB 小灯，通过控制 R、G、B 这三种颜色的亮度就可以显示出各种各样的色彩。那该如何控制 R、G、B 这三种颜色的显示亮度呢？一般一个 R、G、B 这三部分分别使用 8bit 的数据，那么一个像素点就是 8bit*3=24bit，也就是说一个像素点 3 个字节，这种像素格式称为 RGB888。如果在加入 8bit 的 Alpha(透明)通道的话一个像素点就

是 32bit，也就是 4 个字节，这种像素格式称为 ARGB8888。如果学习过 STM32 的话应该还听过 RGB565 这种像素格式，在本章实验中我们使用 ARGB8888 这种像素格式，一个像素占用 4 个字节的内存，这四个字节每个位的分配如图 24.1.1.2 所示：

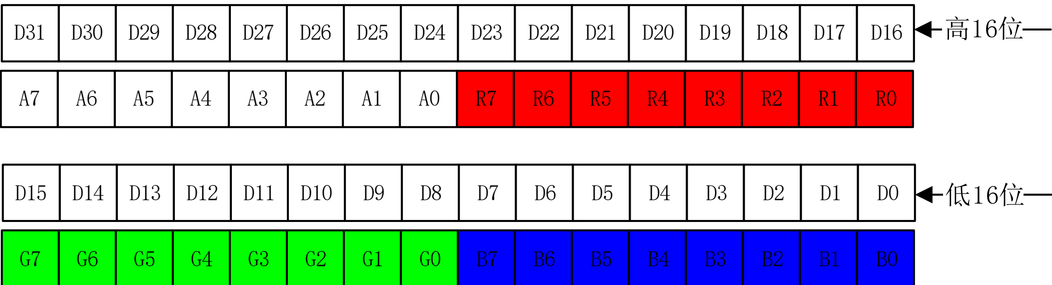


图 24.1.1.2 ARGB8888 数据格式

在图 24.1.1.2 中，一个像素点是 4 个字节，其中 bit31~bit24 是 Alpha 通道，bit23~bit16 是 RED 通道，bit15~bit14 是 GREEN 通道，bit7~bit0 是 BLUE 通道。所以红色对应的值就是 0X00FF0000，蓝色对应的值就是 0X0000FF00，绿色对应的值为 0X000000FF。通过调节 R、G、B 的比例可以产生其它的颜色，比如 0X00FFFF00 就是黄色，0X00000000 就是黑色，0X00FFFFFF 就是白色。大家可以打开电脑的“画图”工具，在里面使用调色板即可获取到想要的颜色对应的数值，如图 24.1.1.3 所示：



图 24.1.1.3 颜色选取

3、LCD 屏幕接口

LCD 屏幕或者说显示器有很多种接口，比如在显示器上常见的 VGA、HDMI、DP 等等，但是 I.MX6U-ALPHA 开发板不支持这些接口。I.MX6U-ALPHA 支持 RGB 接口的 LCD，RGBLCD 接口的信号线如表 24.1.1.1 所示：

信号线	描述
R[7:0]	8 根红色数据线。
G[7:0]	8 根绿色数据线。
B[7:0]	8 根蓝色数据线。
DE	数据使能线。

VSYNC	垂直同步信号线。
HSYNC	水平同步信号线。
PCLK	像素时钟信号线。

表 24.1.1.1 RGB 数据线

表 24.1.1.1 就是 RGBLCD 的信号线，R[7:0]、G[7:0]和 B[7:0]这 24 根是数据线，DE、VSYNC、HSYNC 和 PCLK 这四根是控制信号线。RGB LCD 一般有两种驱动模式：DE 模式和 HV 模式，这两个模式的区别是 DE 模式需要用到 DE 信号线，而 HV 模式不需要用到 DE 信号线，在 DE 模式下是可以不需要 HSYNC 信号线的，即使不接 HSYNC 信号线 LCD 也可以正常工作。

ALIENTEK 一共有三款 RGB LCD 屏幕，型号分别为：ATK-4342(4.3 寸，480*272)、ATK-7084(7 寸，800*480)和 ATK-7016(7 寸，1024*600)，本教程就以 ATK-7016 这款屏幕为例讲解，ATK-7016 的屏幕接口原理图如图 24.1.1.4 所示：

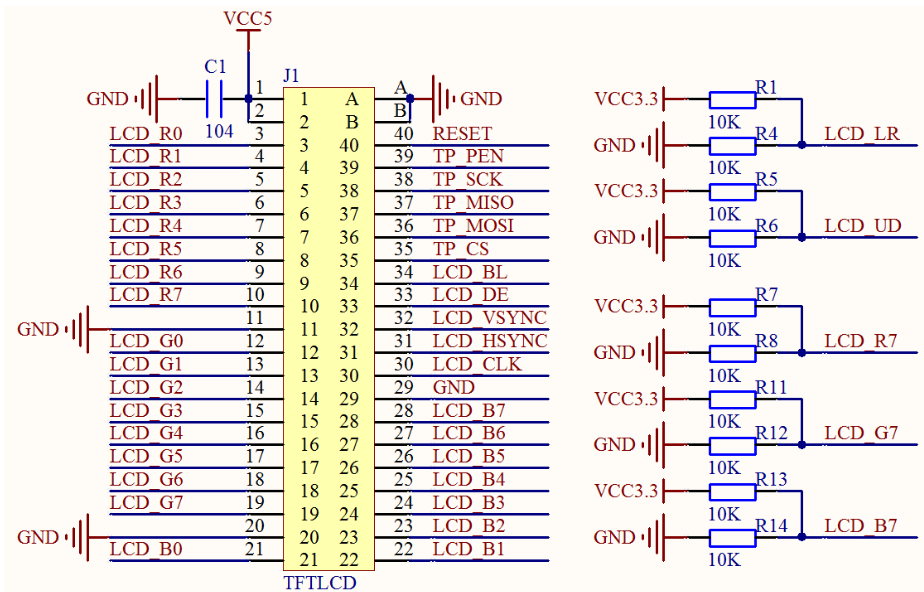


图 24.1.1.4 RGB LCD 液晶屏屏幕接口

图中 J1 就是对外接口，是一个 40PIN 的 FPC 座（0.5mm 间距），通过 FPC 线，可以连接到 I.MX6U-ALPHA 开发板上面，从而实现和 I.MX6U 的连接。该接口十分完善，采用 RGB888 格式，并支持 DE&HV 模式，还支持触摸屏和背光控制。右侧的几个电阻，并不是都焊接的，而是可以用户自己选择。默认情况，R1 和 R6 焊接，设置 LCD_LR 和 LCD_UD，控制 LCD 的扫描方向，是从左到右，从上到下（横屏看）。而 LCD_R7/G7/B7 则用来设置 LCD 的 ID，由于 RGBLCD 没有读写寄存器，也就没有所谓的 ID，这里我们通过在模块上面，控制 R7/G7/B7 的上/下拉，来自定义 LCD 模块的 ID，帮助 MCU 判断当前 LCD 面板的分辨率和相关参数，以提高程序兼容性。这几个位的设置关系如表 24.1.1.2 所示：

M2 LCD_G7	M1 LCD_G7	M0 LCD_R7	LCD ID	说明
0	0	0	4342	ATK-4342 RGBLCD 模块，分辨率：480*272
0	0	1	7084	ATK-7084 RGBLCD 模块，分辨率：800*480
0	1	0	7016	ATK-7016，RGBLCD 模块，分辨率：1024*600
0	1	1	7018	ATK-7018，RGBLCD 模块，分辨率：1280*800
X	X	X	NC	暂时未用到

表 24.1.1.2 ALIENTEK RGBLCD 模块 ID 对应关系

ATK-7016 模块，就设置 M2:M0=010 即可。这样，我们在程序里面，读取 LCD_R7/G7/B7，得到 M0:M2 的值，从而判断 RGBLCD 模块的型号，并执行不同的配置，即可实现不同 LCD 模块的兼容。

4、LCD 时间参数

如果将 LCD 显示一帧图像的过程想象成绘画，那么在显示的过程中就是用一根“笔”在不同的像素点画上不同的颜色。这根笔按照从左至右、从上到下的顺序扫描每个像素点，并且在像素画上对应的颜色，当画到最后一个像素点的时候一幅图像就绘制好了。假如一个 LCD 的分辨率为 1024*600，那么其扫描如图 24.1.1.5 所示：

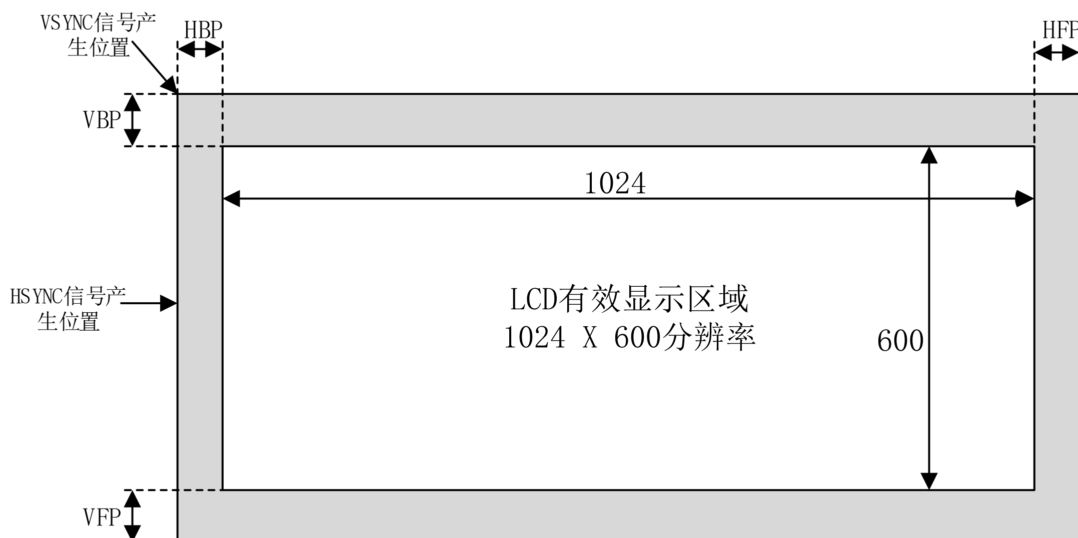


图 24.1.1.5 LCD 一帧图像扫描图

结合图 24.1.1.4 我们来看一下 LCD 是怎么扫描显示一帧图像的。一帧图像也是由一行一行组成的。HSYNC 是水平同步信号，也叫做行同步信号，当产生此信号的话就表示开始显示新的一行了，所以此信号都是在图 24.1.1.5 的最左边。当 VSYNC 信号是垂直同步信号，也叫做帧同步信号，当产生此信号的话就表示开始显示新的一帧图像了，所以此信号在图 24.1.1.4 的左上角。

在图 24.1.1.5 可以看到有一圈“黑边”，真正有效的显示区域是中间的白色部分。那这一圈“黑边”是什么东西呢？这就要从显示器的“祖先”CRT 显示器开始说起了，CRT 显示器就是以前很常见的那种大屁股显示器，在 2019 年应该很少见了，如果在农村应该还是可以见到的。CRT 显示器屁股后面是个电子枪，这个电子枪就是我们上面说的“画笔”，电子枪打出的电子撞击到屏幕上的荧光物质使其发光。只要控制电子枪从左到右扫打万一行(也就是扫描一行)，然后从上到下扫描完所有行，这样一帧图像就显示出来了。也就是说，显示一帧图像电子枪是按照‘Z’形在运动，当扫描速度很快的时候看起来就是一幅完成的画面了。

当显示完一行以后会发出 HSYNC 信号，此时电子枪就会关闭，然后迅速的移动到屏幕的左边，当 HSYNC 信号结束以后就可以显示新的一行数据了，电子枪就会重新打开。在 HSYNC 信号结束到电子枪重新打开之间会插入一段延时，这段延时就图 24.1.1.5 中的 HBP。当显示完一行以后就会关闭电子枪等待 HSYNC 信号产生，关闭电子枪到 HSYNC 信号产生之间会插入一段延时，这段延时就是图 24.1.1.5 中的 HFP 信号。同理，当显示完一帧图像以后电子枪也会关闭，然后等到 VSYNC 信号产生，期间也会加入一段延时，这段延时就是图 24.1.1.5 中的 VFP。VSYNC 信号产生，电子枪移动到左上角，当 VSYNC 信号结束以后电子枪重新打开，中间也会加入一段延时，这段延时就是图 24.1.1.5 中的 VBP。

HBP、HFP、VBP 和 VFP 就是导致图 24.1.1.5 中黑边的原因，但是这是 CRT 显示器存在黑边的原因，现在是 LCD 显示器，不需要电子枪了，那么为何还会有黑边呢？这是因为 RGB LCD 屏幕内部是有一个 IC 的，发送一行或者一帧数据给 IC，IC 是需要反应时间的。通过这段反应时间可以让 IC 识别到一行数据扫描完了，要换行了，或者一帧图像扫描完了，要开始下一帧图像显示了。因此，在 LCD 屏幕中继续存在 HBP、HFP、VPB 和 VFP 这四个参数的主要目的是为了锁定有效的像素数据。这四个时间是 LCD 重要的时间参数，后面编写 LCD 驱动的时候要用到的，至于这四个时间参数具体值是多少，那要需要去查看所使用的 LCD 数据手册了。

5、RGB LCD 屏幕时序

上面讲了行显示和帧显示，我们来看一下行显示对应的时序图，如图 24.1.1.6 所示：

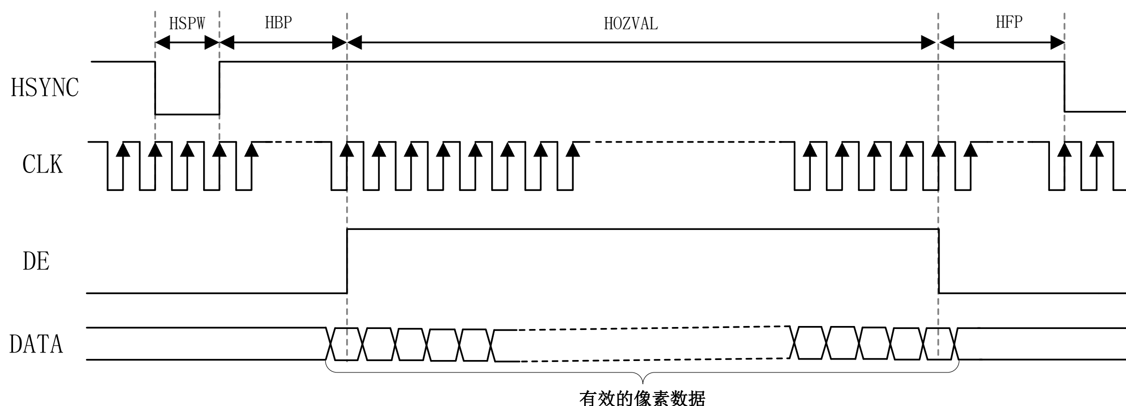


图 24.1.1.6 行显示时序

图 24.1.1.6 就是 RGB LCD 的行显示时序，我们来分析一下其中重要的几个参数：

HSYNC：行同步信号，当此信号有效的话就表示开始显示新的一行数据，查阅所使用的 LCD 数据手册可以知道此信号是低电平有效还是高电平有效，假设此时是低电平有效。

HSPW：有些地方也叫做 thp，是 HSYNC 信号宽度，也就是 HSYNC 信号持续时间。HSYNC 信号不是一个脉冲，而是需要持续一段时间才是有效的，单位为 CLK。

HBP：有些地方叫做 thb，前面已经讲过了，术语叫做行同步信号后肩，单位是 CLK。

HOZVAL：有些地方叫做 thd，显示一行数据所需的时间，假如屏幕分辨率为 1024*600，那么 HOZVAL 就是 1024，单位为 CLK。

HFP：有些地方叫做 thf，前面已经讲过了，术语叫做行同步信号前肩，单位是 CLK。

当 HSYNC 信号发出以后，需要等待 HSPW+HBP 个 CLK 时间才会接收到真正有效的像素数据。当显示完一行数据以后需要等待 HFP 个 CLK 时间才能发出下一个 HSYNC 信号，所以显示一行所需要的时间就是：HSPW + HBP + HOZVAL + HFP。

一帧图像就是由很多个行组成的，RGB LCD 的帧显示时序如图 24.1.1.7 所示：

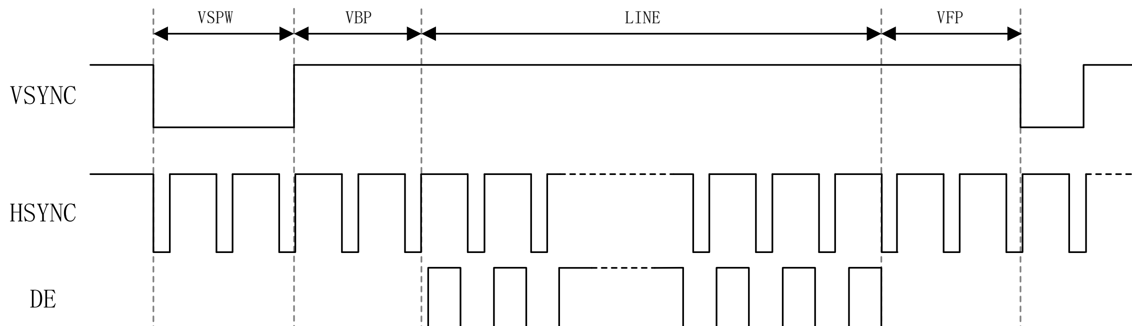


图 24.1.1.7 帧显示时序图

图 24.1.1.7 就是 RGB LCD 的帧显示时序，我们来分析一下其中重要的几个参数：

VSYNC：帧同步信号，当此信号有效的话就表示开始显示新的一帧数据，查阅所使用的LCD 数据手册可以知道此信号是低电平有效还是高电平有效，假设此时是低电平有效。

VSPW：些地方也叫做 tvp，是 VSYNC 信号宽度，也就是 VSYNC 信号持续时间，单位为1 行的时间。

VBP：有些地方叫做 tvb，前面已经讲过了，术语叫做帧同步信号后肩，单位为 1 行的时间。

LINE：有些地方叫做 tvd，显示一帧有效数据所需的时间，假如屏幕分辨率为 1024*600，那么 LINE 就是 600 行的时间。

VFP：有些地方叫做 tvf，前面已经讲过了，术语叫做帧同步信号前肩，单位为 1 行的时间。

显示一帧所需要的时间就是：VSPW+VBP+LINE+VFP 个行时间，最终的计算公式：

T = (VSPW+VBP+LINE+VFP) * (HSPW + HBP + HOZVAL + HFP)

因此我们在配置一款 RGB LCD 的时候需要知道这几个参数：HOZVAL(屏幕有效宽度)、LINE(屏幕有效高度)、HBP、HSPW、HFP、VSPW、VBP 和 VFP。ALIENTEK 三款 RGB LCD 屏幕的参数如表 24.1.1.3 所示：

屏幕型号	参数	值	单位
ATK4342	水平显示区域	480	tCLK
	HSPW(thp)	1	tCLK
	HBP(thb)	40	tCLK
	HFP(thf)	5	tCLK
	垂直显示区域	272	th
	VSPW(tvp)	1	th
	VBP(tvb)	8	th
	VFP(tvf)	8	th
	像素时钟	9	MHz
ATK4384	水平显示区域	800	tCLK
	HSPW(thp)	48	tCLK
	HBP(thb)	88	tCLK
	HFP(thf)	40	tCLK
	垂直显示区域	480	th
	VSPW(tvp)	3	th
	VBP(tvb)	32	th
	VFP(tvf)	13	th
	像素时钟	31	MHz
ATK7084	水平显示区域	800	tCLK
	HSPW(thp)	1	tCLK
	HBP(thb)	46	tCLK
	HFP(thf)	210	tCLK
	垂直显示区域	480	th
	VSPW(tvp)	1	th
	VBP(tvb)	23	th
	VFP(tvf)	22	th
	像素时钟	33.3	MHz

ATK7016	水平显示区域	1024	tCLK
	HSPW(thp)	20	tCLK
	HBP(thb)	140	tCLK
	HFP(thf)	160	tCLK
	垂直显示区域	600	th
	VSPW(tvp)	3	th
	VBP(tvb)	20	th
	VFP(tvf)	12	th
	像素时钟	51.2	MHz

表 24.1.1.3 RGB LCD 屏幕时间参数

6、像素时钟

像素时钟就是 RGB LCD 的时钟信号，以 ATK7016 这款屏幕为例，显示一帧图像所需要的时钟数就是：

= (VSPW+VBP+LINE+VFP) * (HSPW + HBP + HOZVAL + HFP)

= (3 + 20 + 600 + 12) * (20 + 140 + 1024 + 160)

= 635 * 1344

= 853440。

显示一帧图像需要 853440 个时钟数，那么显示 60 帧就是：853440 * 60 = 51206400≈51.2M，所以像素时钟就是 51.2MHz。

I.MX6U 的 eLCDIF 接口时钟图如图 24.1.1.8 所示：

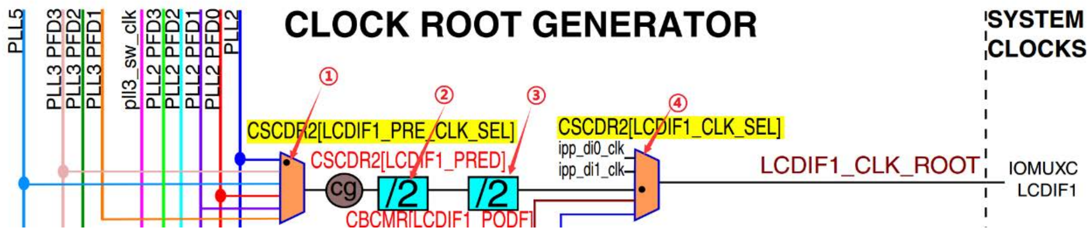


图 24.1.1.8 LCDIF 接口时钟图

①、此部分是一个选择器，用于选择哪个 PLL 可以作为 LCDIF 时钟源，由寄存器 CCM_CSCDR2 的位 LCDIF1_PRE_CLK_SEL(bit17:15)来决定，LCDIF1_PRE_CLK_SEL 选择设置如表 24.1.1.4 所示：

值	时钟源
0	PLL2 作为 LCDIF 的时钟源。
1	PLL3_PFD3 作为 LCDIF 的时钟源。
2	PLL5 作为 LCDIF 的时钟源。
3	PLL2_PFD0 作为 LCDIF 的时钟源。
4	PLL2_PFD1 作为 LCDIF 的时钟源。
5	PLL3_PFD1 作为 LCDIF 的时钟源。

表 24.1.1.4　LCDIF 时钟源选择

在第 16 章讲解 I.MX6U 时钟系统的时候说过有个专用的 PLL5 给 VIDEO 使用，所以 LCDIF1_PRE_CLK_SEL 设置为 2。

②、此部分是 LCDIF 时钟的预分频器，由寄存器 CCM_CSCDR2 的位 LCDIF1_PRED 来决定预分频值。可设置值为 0~7，分别对应 1~8 分频。

③、此部分进一步分频，由寄存器 CBCMR 的位 LCDIF1_PODF 来决定分频值。可设置值为 0~7，分别对应 1~8 分频。

④、此部分是一个选择器，选择 LCDIF 最终的根时钟，由寄存器 CSCDR2 的位 LCDIF1_CLK_SEL 决定，LCDIF1_CLK_SEL 选择设置如表 24.1.1.5 所示：

值	时钟源
0	前面复用器出来的时钟，也就是前面 PLL5 出来的时钟作为 LCDIF 的根时钟。
1	ipp_di0_clk 作为 LCDIF 的根时钟。
2	ipp_di1_clk 作为 LCDIF 的根时钟。
3	ldb_di0_clk 作为 LCDIF 的根时钟。
4	ldb_di1_clk 作为 LCDIF 的根时钟。

表 24.1.1.4 LCDIF 根时钟选择

这里肯定选择 PLL5 出来的那一路时钟作为 LCDIF 的根时钟，因此 LCDIF1_CLK_SEL 设置为 0。LCDIF 既然选择了 PLL5 作为时钟源，那么还需要初始化 PLL5，LCDIF 的时钟是由 PLL5 和图 24.1.1.8 中的②、③这两个分频值决定的，所以需要对这三个进行合理的设置以搭配出所需的时钟值，我们就以 ATK7016 屏幕所需的 51.2MHz 为例，看看如何进行配置。

PLL5 频率设置涉及到四个寄存器：CCM_PLL_VIDEO、CCM_PLL_VIDEO_NUM、CCM_PLL_VIDEO_DENOM、CCM_MISC2。其中 CCM_PLL_VIDEO_NUM 和 CCM_PLL_VIDEO_DENOM 这两个寄存器是用于小数分频的，我们这里为了简单不使用小数分频，因此这两个寄存器设置为 0。

PLL5 的时钟计算公式如下：

PLL5_CLK = OSC24M * (loopDivider + (denominator / numerator)) / postDivider

不使用小数分频的话 PLL5 时钟计算公式就可以简化为：

PLL5_CLK = OSC24M * loopDivider / postDivider

OSC24M 就是 24MHz 的有源晶振，现在的问题就是设置 loopDivider 和 postDivider。先来看一下寄存器 CCM_PLL_VIDEO，此寄存器结构如图 24.1.1.9 所示：

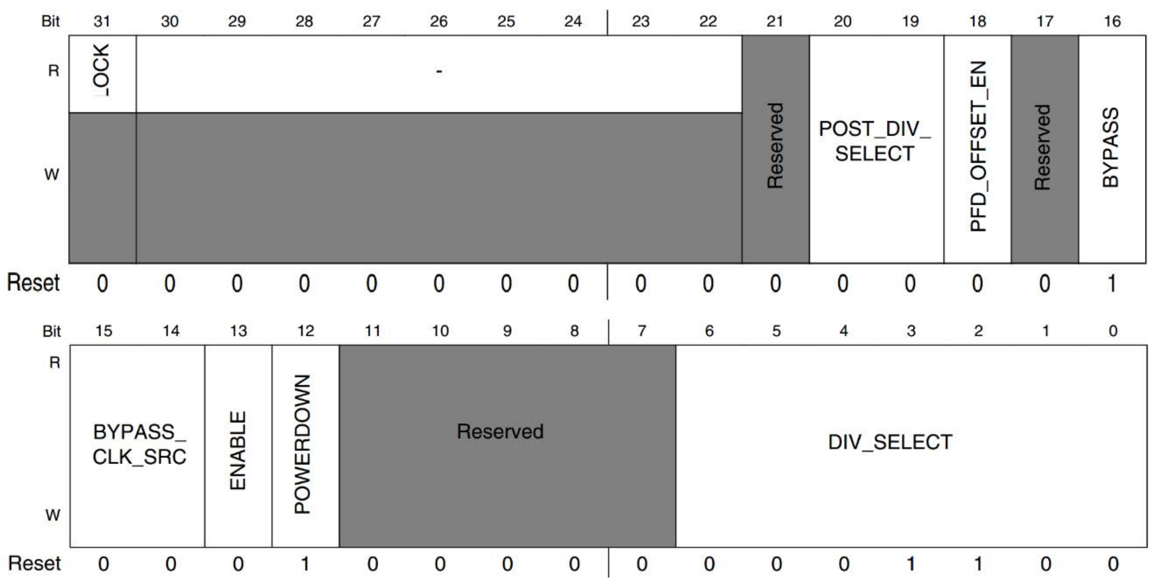


图 24.1.1.9 寄存器 CCM_PLL_VIDEO 结构

寄存器 CCM_PLL_VIDEO 用到的重要的位如下：

POST_DIV_SLECT(bit20:19): 此位和寄存器 CCM_ANALOG_CCMSC2 的 VIDEO_DIV 位

共同决定了 postDivider，为 0 的话是 4 分频，为 1 的话是 2 分频，为 2 的话是 1 分频。本章设置为 2，也就是 1 分频。

ENABLE(bit13)：PLL5(PLL_VIDEO)使能为，为 1 的话使能 PLL5，为 0 的话关闭 PLL5。

DIV_SELECT(bit6:0)：loopDivider 值，范围为 27~54，本章设置为 32。

寄存器 CCM_ANALOG_MISC2 的位 VIDEO_DIV(bit31:30)与寄存器 CCM_PLL_VIDEO 的位 POST_DIV_SLECT(bit20:19)共同决定了 postDivider，通过这两个的配合可以获得 2、4、8、16 分频。本章将 VIDEO_DIV 设置为 0，也就是 1 分频，因此 postDivider 就是 1，loopDivider 设置为 32，PLL5 的时钟频率就是：

PLL5_CLK = OSC24M * loopDivider / postDivider
= 24M * 32 / 1
= 768MHz。

PLL5 此时为 768MHz，在经过图 24.1.1.8 中的②和③进一步分频，设置②中为 3 分频，也就是寄存器 CCM_CSCDR2 的位 LCDIF1_PRED(bit14:12)为 2。设置③中为 5 分频，就是寄存器 CCM_CBCMR 的位 LCDIF1_PODF(bit25:23)为 4。设置好以后最终进入到 LCDIF 的时钟频率就是：768/3/5 =51.2MHz，这就是我们需要的像素时钟频率。

7、显存

在讲像素格式的时候就已经说过了，如果采用 ARGB8888 格式的话一个像素需要 4 个字节的内存来存放像素数据，那么 1024*600 分辨率就需要 1024*600*4=2457600B≈2.4MB 内存。但是 RGB LCD 内部是没有内存的，所以就需要在开发板上的 DDR3 中分出一段内存作为 RGB LCD 屏幕的显存，我们如果要在屏幕上显示什么图像的话直接操作这部分显存即可。

24.1.2 eLCDIF 接口

eLCDIF 是 I.MX6U 自带的液晶屏幕接口，用于连接 RGB LCD 接口的屏幕，eLCDIF 接口特性如下：

、支持 RGB LCD 的 DE 模式。
、支持 VSYNC 模式以实现高速数据传输。
、支持 ITU-R BT.656 格式的 4:2:2 的 YCbCr 数字视频，并且将其转换为模拟 TV 信号。
、支持 8/16/18/24/32 位 LCD。

eLCDIF 支持三种接口：MPU 接口、VSYNC 接口和 DOTCLK 接口，这三种接口区别如下：

1、MPU 接口

MPU 接口用于在 I.MX6U 和 LCD 屏幕直接传输数据和命令，这个接口用于 6080/8080 接口的 LCD 屏幕，比如我们学习 STM32 的时候常用到的 MCU 屏幕。如果寄存器 LCDIF_CTRL 的位 DOTCLK_MODE、DVI_MODE 和 VSYNC_MODE 都为 0 的话就表示 LCDIF 工作在 MPU 接口模式。关于 MPU 接口的详细信息以及时序参考《I.MX6ULL 参考手册》第 2150 页的“34.4.6 MPU Interface”小节，本教程不使用 MPU 接口。

2、VSYNC 接口

VSYNC 接口时序和 MPU 接口时序基本一样，只是多了 VSYNC 信号来作为帧同步，当 LCDIF_CTRL 的位 VSYNC_MODE 为 1 的时候此接口使能。关于 VSYNC 接口的详细信息请参考《I.MX6ULL 参考手册》第 2152 页的“34.4.7 VSYNC Interface”小节，本教程不使用 VSYNC 接口。

3、DOTCLK 接口

DOTCLK 接口就是用来连接 RGB LCD 接口屏幕的， 它包括 VSYNC、HSYNC、DOTCLK 和 ENABLE(可选的)这四个信号，这样的接口通常被成为 RGB 接口。DOTCLK 接口时序如图 24.1.2.1 所示：

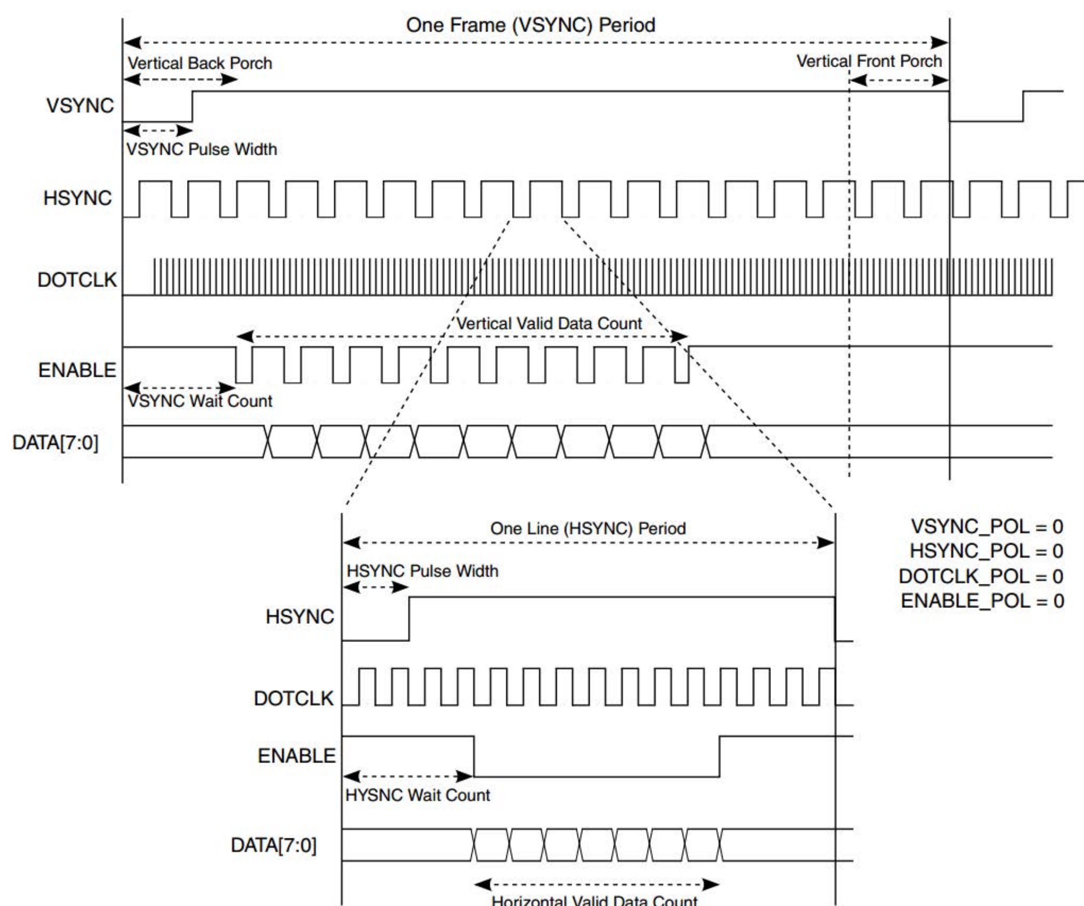


图 24.1.2.1 DOTCLK 接口时序

图 24.1.2.1 是不是和图 24.1.1.6、图 24.1.1.7 很类似，因为 DOTCLK 接口就是连接 RGB 屏幕的，本教程使用的就是 DOTCLK 接口。

eLCDIF 要驱动起来 RGB LCD 屏幕，重点是配置好上一小节讲解的那些时间参数即可，这个通过配置相应的寄存器就可以了，所以我们接下来看一下 eLCDIF 接口的几个重要的寄存器，首先看一下 LCDIF_CTRL 寄存器，此寄存器结构如图 24.1.2.1 所示：

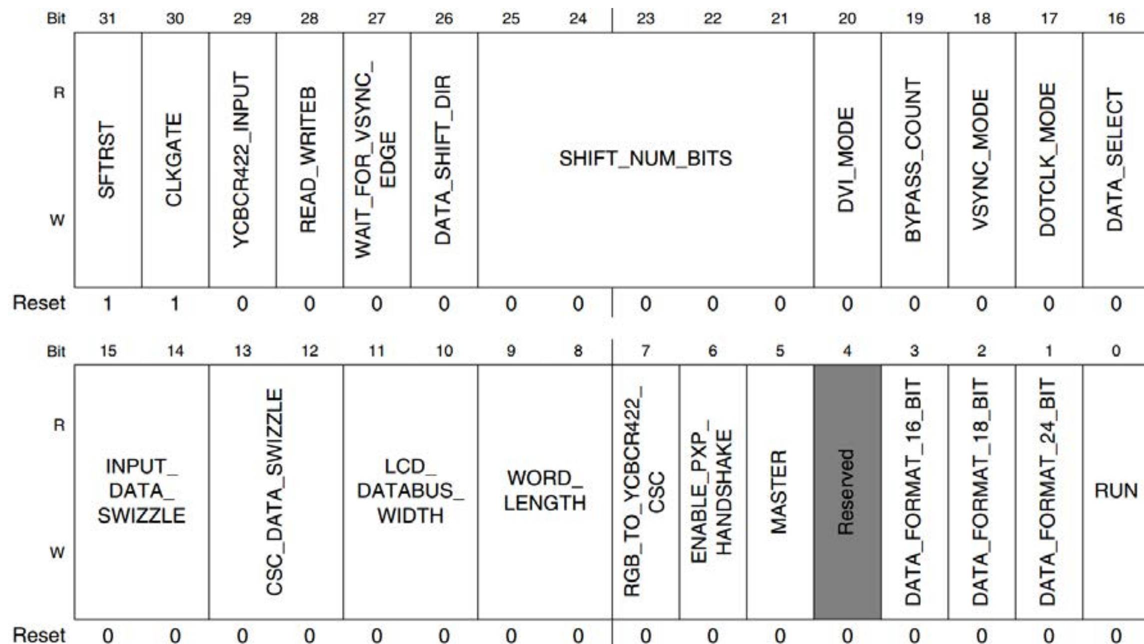


图 24.1.2.1 寄存器 LCDIF_CTRL 结构

寄存器 LCDIF_CTRL 用到的重要位如下：

SFTRST(bit31)：eLCDIF 软复位控制位，当此位为 1 的话就会强制复位 LCD。

CLKGATE(bit30)：正常运行模式下，此位必须为 0！如果此位为 1 的话时钟就不会进入到 LCDIF。

BYPASS_COUNT(bit19)：如果要工作在 DOTCLK 模式的话就此位必须为 1。

VSYNC_MODE(bit18)：此位为 1 的话 LCDIF 工作在 VSYNC 接口模式。

DOTCLK_MODE(bit17)：此位为 1 的话 LCDIF 工作在 DOTCLK 接口模式。

INPUT_DATA_SWIZZLE(bit15:14)：输入数据字节交换设置，此位为 0 的话不交换字节也就是小端模式；为 1 的话交换所有字节，也就是大端模式；为 2 的话半字交换；为 3 的话在每个半字内进行字节交换。本章我们设置为 0，也就是不使用字节交换。

CSC_DATA_SWIZZLE(bit13:12)：CSC 数据字节交换设置，交换方式和 INPUT_DATA_SWIZZLE 一样，本章设置为 0，不使用字节交换。

LCD_DATABUS_WIDTH(bit11:10)：LCD 数据总线宽度，为 0 的话总线宽度为 16 位；为 1 的话总线宽度为 8 位；为 2 的话总线宽度为 18 位；为 3 的话总线宽度为 24 位。本章我们使用 24 位总线宽度。

WORD_LENGTH(bit9:8)：输入的数据格式，也就是像素数据宽度，为 0 的话每个像素 16 位；为 1 的话每个像素 8 位；为 2 的话每个像素 18 位；为 3 的话每个像素 24 位。

MASTER(bit5)：为 1 的话设置 eLCDIF 工作在主模式。

DATA_FORMAT_16_BIT(bit3)：当此位为 1 并且 WORD_LENGTH 为 0 的时候像素格式为 ARGB555，当此位为 0 并且 WORD_LENGTH 为 0 的时候像素格式为 RGB565。

DATA_FORMAT_18_BIT(bit2)：只有当 WORD_LENGTH 为 2 的时候此位才有效，此位为 0 的话低 18 位有效，像素格为 RGB666，高 14 位数据无效。当此位为 1 的话高 18 位有效，像素格式依旧是 RGB666，但是低 14 位数据无效。

DATA_FORMAT_24_BIT(bit1)：只有当 WORD_LENGTH 为 3 的时候此位才有效，为 0 的时候表示全部的 24 位数据都有效。为 1 的话实际输入的数据有效位只有 18 位，虽然输入的是 24 位数据，但是每个颜色通道的高 2 位数据会被丢弃掉。

RUN(bit0)：eLCDIF 接口运行控制位，当此位为 1 的话 eLCDIF 接口就开始传输数据，也就是 eLCDIF 的使能位。

接下来看一下寄存器 LCDIF_CTRL1，此寄存器我们只用到位 BYTE_PACKING_FORMAT(bit19:16)，此位用来决定在 32 位的数据中哪些字节的数据有效，默认值为 0XF，也就是所有的字节有效，当为 0 的话表示所有的字节都无效。如果显示的数据是 24 位(ARGB 格式，但是 A 通道不传输)的话就设置此位为 0X7。

接下来看一下寄存器 LCDIF_TRANSFER_COUNT，这个寄存器用来设置所连接的 RGB LCD 屏幕分辨率大小，此寄存器结构如图 24.1.2.2 所示：

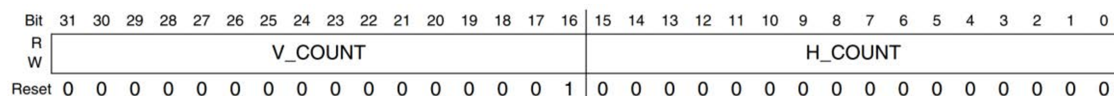


图 24.1.2.2 寄存器 LCDIF_TRANSFER_COUNT 结构

寄存器 LCDIF_TRANSFER_COUNT 分为两部分，高 16 位和低 16 位，高 16 位是 V_COUNT，是 LCD 的垂直分辨率。低 16 位是 H_COUNT，是 LCD 的水平分辨率。如果 LCD 分辨率为 1024*600 的话，那么 V_COUNT 就是 600，H_COUNT 就是 1024。

接下来看一下寄存器 LCDIF_VDCTRL0，这个寄存器是 VSYNC 和 DOTCLK 模式控制寄存器 0，寄存器结构如图 24.1.2.3 所示：

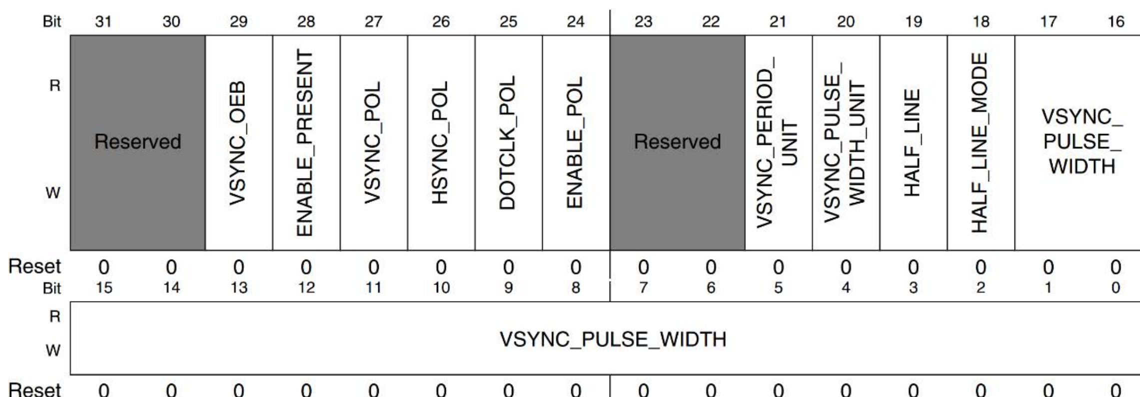


图 24.1.2.3 寄存器 LCDIF_VDCTRL0 结构

寄存器 LCDIF_VDCTRL0 用到的重要位如下：

VSYNC_OEB(bit29)：VSYNC 信号方向控制位，为 0 的话 VSYNC 是输出，为 1 的话 VSYNC 是输入。

ENABLE_PRESENT(bit28)：EBABLE 数据线使能位，也就是 DE 数据线。为 1 的话使能 ENABLE 数据线，为 0 的话关闭 ENABLE 数据线。

VSYNC_POL(bit27)：VSYNC 数据线极性设置位，为 0 的话 VSYNC 低电平有效，为 1 的话 VSYNC 高电平有效，要根据所使用的 LCD 数据手册来设置。

HSYNC_POL(bit26)：HSYNC 数据线极性设置位，为 0 的话 HSYNC 低电平有效，为 1 的话 HSYNC 高电平有效，要根据所使用的 LCD 数据手册来设置。

DOTCLK_POL(bit25)：DOTCLK 数据线(像素时钟线 CLK) 极性设置位，为 0 的话下降沿锁存数据，上升沿捕获数据，为 1 的话相反，要根据所使用的 LCD 数据手册来设置。

ENABLE_POL(bit24)：EANBLE 数据线极性设置位，为 0 的话低电平有效，为 1 的话高电平有效。

VSYNC_PERIOD_UNIT(bit21)：VSYNC 信号周期单位，为 0 的话 VSYNC 周期单位为像素时钟。为 1 的话 VSYNC 周期单位是水平行，如果使用 DOTCLK 模式话就要设置为 1。

VSYNC_PULSE_WIDTH_UNIT(bit20)：VSYNC 信号脉冲宽度单位，和 VSYNC_PERIOD_UNUT 一样，如果使用 DOTCLK 模式的话要设置为 1。

VSYNC_PULSE_WIDTH(bit17:0)：VSPW 参数设置位。

接下来看一下寄存器 LCDIF_VDCTRL1，这个寄存器是 VSYNC 和 DOTCLK 模式控制寄存器 1，此寄存器只有一个功能，用来设置 VSYNC 总周期，就是：屏幕高度+VSPW+VBP+VFP。

接下来看一下寄存器 LCDIF_VDCTRL2，这个寄存器分为高 16 位和低 16 位两部分，高 16 位是 HSYNC_PULSE_WIDTH，用来设置 HSYNC 信号宽度，也就是 HSPW。低 16 位是 HSYNC_PERIOD，设置 HSYNC 总周期，就是：屏幕宽度+HSPW+HBP+HFP。

接下来看一下寄存器 LCDIF_VDCTRL3，此寄存器结构如图 24.1.2.4 所示：

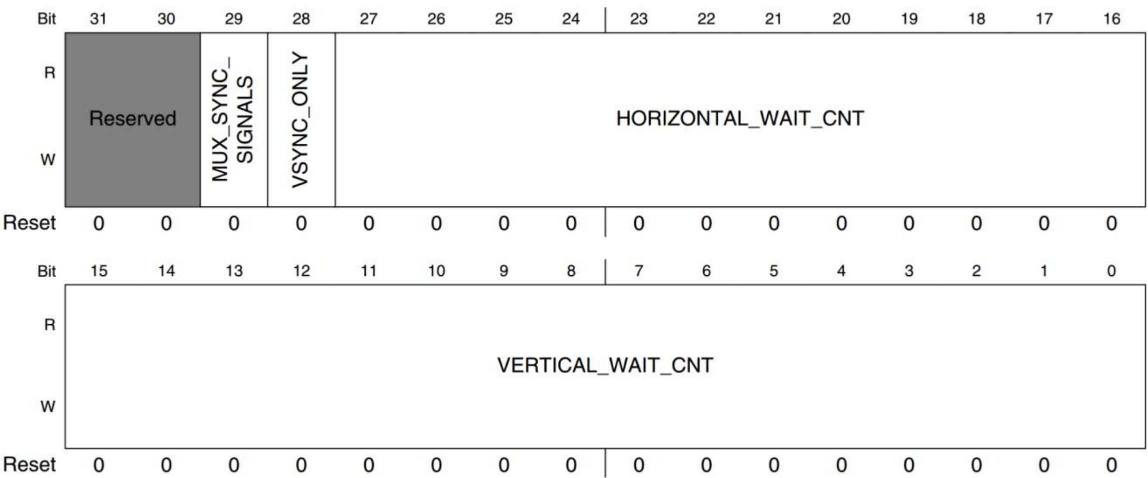


图 24.1.2.4 寄存器 LCDIF_VDCTRL3 结构

寄存器 LCDIF_VDCTRL3 用到的重要位如下：

HORIZONTAL_WAIT_CNT(bit27:16)：此位用于 DOTCLK 模式，用于设置 HSYNC 信号产生到有效数据产生之间的时间，也就是 HSPW+HBP。

VERTICAL_WAIR_CNT(bit15:0)：和 HORIZONTAL_WAIT_CNT 一样，只是此位用于 VSYNC 信号，也就是 VSPW+VBP。

接下来看一下寄存器 LCDIF_VDCTRL4，此寄存器结构如图 24.1.2.5 所示：

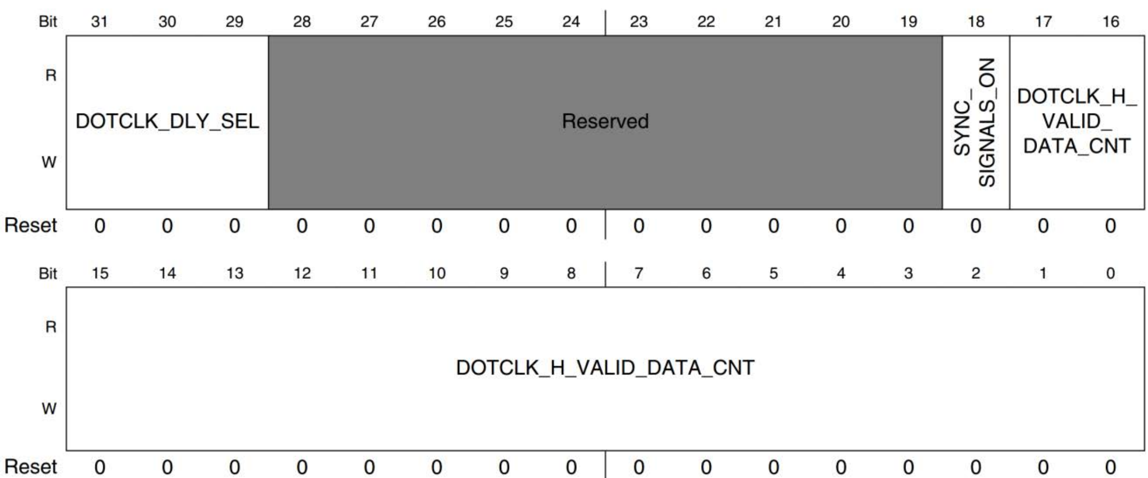


图 24.1.2.5 寄存器 LCDIF_VDCTRL4 结构

寄存器 LCDIF_VDCTRL4 用到的重要位如下：

SYNC_SIGNALS_ON(bit18)：同步信号使能位，设置为 1 的话使能 VSYNC、HSYNC、

DOTCLK 这些信号。

DOTCLK_H_VALID_DATA_CNT(bit15:0)： 设置 LCD 的宽度，也就是水平像素数量。

最后在看一下寄存器 LCDIF_CUR_BUF 和 LCDIF_NEXT_BUF，这两个寄存器分别为当前帧和下一帧缓冲区，也就是 LCD 显存。一般这两个寄存器保存同一个地址，也就是划分给 LCD 的显存首地址。

关于 eLCDIF 接口的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 2165 页的 34.6 小节。本章我们使用 I.MX6U 的 eLCDIF 接口来驱动 ALIENTEK 的 ATK7016 这款屏幕，配置步骤如下：

1、初始化 LCD 所使用的 IO

首先肯定是初始化 LCD 所示使用的 IO，将其复用为 eLCDIF 接口 IO。

2、设置 LCD 的像素时钟

查阅所使用的 LCD 屏幕数据手册，或者自己计算出的时钟像素，然后设置 CCM 相应的寄存器。

3、配置 eLCDIF 接口

设置 LCDIF 的寄存器 CTRL、CTRL1、TRANSFER_COUNT、VDCTRL0~4、CUR_BUF 和 NEXT_BUF。根据 LCD 的数据手册设置相应的参数。

4、编写 API 函数

驱动 LCD 屏幕的目的就是显示内容，所以需要编写一些基本的 API 函数，比如画点、画线、画圆函数，字符串显示函数等。

24.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、RGB LCD 接口。

③、DDR3

④、eLCDIF

RGB LCD 接口在 I.MX6U-ALPHA 开发板底板上，原理图如图 24.2.1 所示：

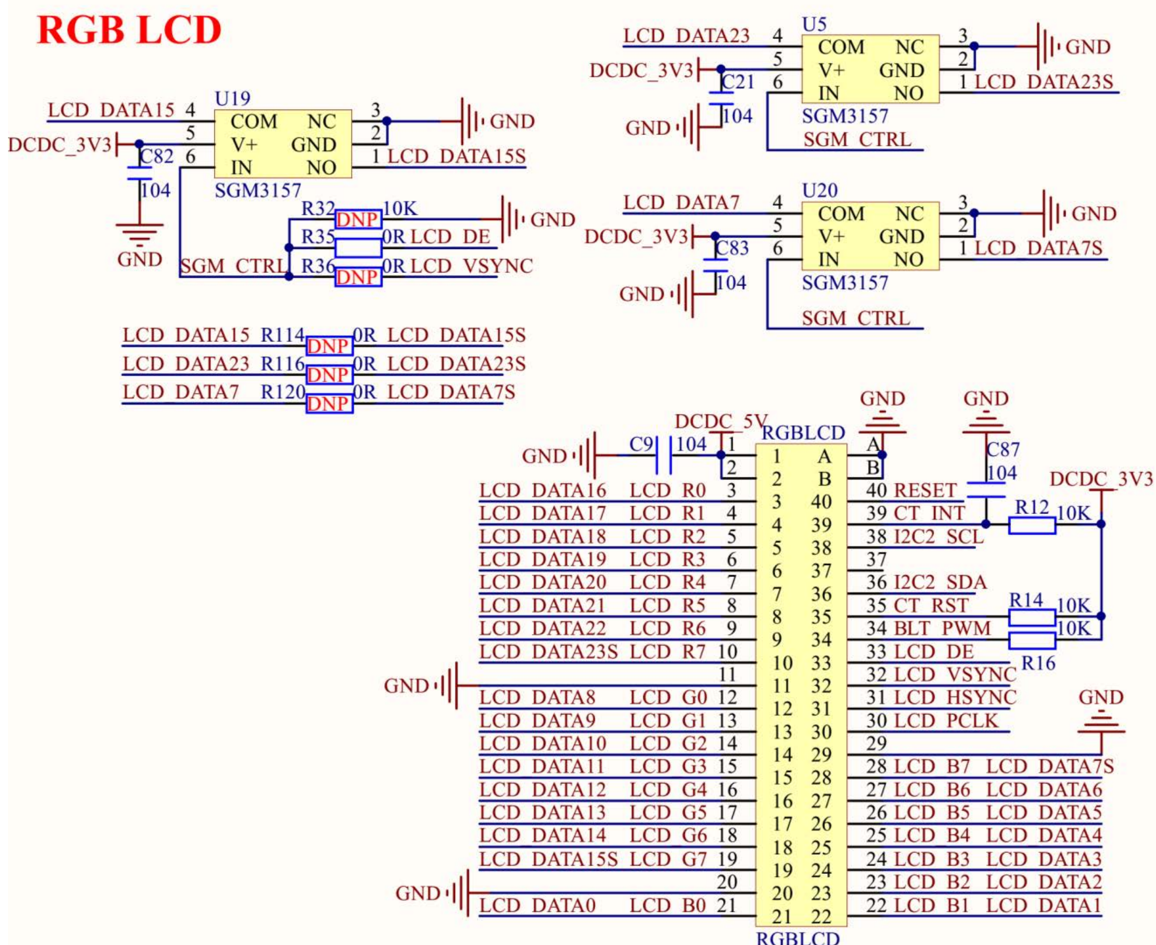


图 24.2.1 RGB LCD 接口原理图

图 24.2.1 中三个 SGM3157 的目的是在未使用 RGBLCD 的时候将 LCD_DATA7、LCD_DATA15 和 LCD_DATA23 这三个线隔离开来，因为 ALIENTEK 的屏幕的 LCD_R7/G7/B7 着几个线用来设置 LCD 的 ID，所以这几根线上有上拉/下拉电阻。但是 I.MX6U 的 BOOT 设置也用到了 LCD_DATA7、LCD_DATA15 和 LCD_DATA23 这三个引脚，所以接上屏幕以后屏幕上的 ID 电阻就会影响到 BOOT 设置，会导致代码无法运行，所以先将其隔离开来，如果要使用 RGB LCD 屏幕的时候再通过 LCD_DE 将其“连接”起来。我们需要 40P 的 FPC 线将 ATK7016 屏幕和 I.MX6U-ALPHA 开发板连接起来，如图 24.2.2.2 所示：



图 24.2.2 屏幕和开发板连接图

24.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 15_lcd。

本章实验在上一章例程的基础上完成，更改工程名字为“lcd”，然后在 bsp 文件夹下创建名为“lcd”的文件夹，在 bsp/lcd 中新建 bsp_lcd.c、bsp_lcd.h、bsp_lcdapi.c、bsp_lcdapi.h 和 font.h 这五个文件。bsp_lcd.c 和 bsp_lcd.h 是 LCD 的驱动文件，bsp_lcdapi.c 和 bsp_lcdapi.h 是 LCD 的 API 操作函数文件，font.h 是字符集点阵数据数组文件。在 bsp_lcd.h 中输入如下内容：

示例代码 24.3.1 bsp_lcd.h 文件代码

```
1  #ifndef _BSP_LCD_H
2  #define _BSP_LCD_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_lcd.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : LCD 驱动文件头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/3 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 颜色宏定义 */
16 #define LCD_BLUE          0x000000FF
17 #define LCD_GREEN         0x0000FF00
18 #define LCD_RED           0x00FF0000
19 /* 省略掉其它宏定义，完整的请参考实验例程 */
```

```
20 #define LCD_ORANGE       0x00FFA500
21 #define LCD_TRANSPARENT  0x00000000
22
23 #define LCD_FRAMEBUF_ADDR   (0x89000000)  /* LCD 显存地址 */
24
25 /* LCD 控制参数结构体 */
26 struct tftlcd_typedef{
27     unsigned short height;       /* LCD 屏幕高度          */
28     unsigned short width;        /* LCD 屏幕宽度          */
29     unsigned char pixsize;       /* LCD 每个像素所占字节大小 */
30     unsigned short vspw;         /* VSYNC 信号宽度         */
31     unsigned short vbpd;         /* 帧同步信号后肩          */
32     unsigned short vfpd;         /* 帧同步信号前肩          */
33     unsigned short hspw;         /* HSYNC 信号宽度         */
34     unsigned short hbpd;         /* 水平同步信号后见肩       */
35     unsigned short hfpd;         /* 水平同步信号前肩         */
36     unsigned int framebuffer;    /* LCD 显存首地址          */
37     unsigned int forecolor;      /* 前景色                */
38     unsigned int backcolor;      /* 背景色                */
39 };
40
41 extern struct tftlcd_typedef tftlcd_dev;
42
43 /* 函数声明 */
44 void lcd_init(void);
45 void lcdgpio_init(void);
46 void lcdclk_init(unsigned char loopDiv, unsigned char prediv,
                    unsigned char div);
47 void lcd_reset(void);
48 void lcd_noreset(void);
49 void lcd_enable(void);
50 void video_pllinit(unsigned char loopdivi, unsigned char postdivi);
51 inline void lcd_drawpoint(unsigned short x,unsigned short y,
                            unsigned int color);
52 inline unsigned int lcd_readpoint(unsigned short x,
                                    unsigned short y);
53 void lcd_clear(unsigned int color);
54 void lcd_fill(unsigned short x0, unsigned short y0,
                 unsigned short x1, unsigned short y1,
                 unsigned int color);
55 #endif
```

在文件 bsp_lcd.h 中一开始定义了一些常用的颜色宏定义，颜色格式都是 ARGB8888 的。第 23 行的宏 LCD_FRAMEBUF_ADDR 是显存首地址，此处将显存首地址放到了 0X89000000

地址处。这个要根据所使用的 LCD 屏幕大小和 DDR 内存大小来确定的，前面我们说了 ATK7016 这款 RGB 屏幕所需的显存大小为 2.4MB，而 I.MX6U-ALPHA 开发板配置的 DDR 有 256 和 512MB 两种类型，内存地址范围分别为 0X80000000~0X90000000 和 0X80000000~0XA0000000。所以 LCD 显存首地址选择为 0X89000000 不管是 256MB 还是 512MB 的 DDR 都可以使用。

第 26 行的结构体 tftlcd_typedef 是 RGB LCD 的控制参数结构体，里面包含了跟 LCD 配置有关的一些成员变量。最后就是一些变量和函数是声明。

在 bsp_lcd.c 中输入如下内容：

示例代码 24.3.2 bsp_lcd.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名      : bsp_lcd.c
作者        : 左忠凯
版本        : V1.0
描述        : LCD 驱动文件。
其他        : 无
论坛        : www.openedv.com
日志        : 初版 V1.0 2019/1/3 左忠凯创建
***************************************************************/
1   #include "bsp_lcd.h"
2   #include "bsp_gpio.h"
3   #include "bsp_delay.h"
4   #include "stdio.h"
5   
6   /* 液晶屏参数结构体 */
7   struct tftlcd_typedef tftlcd_dev;
8   
9   /*
10   * @description   : 始化 LCD
11   * @param         : 无
12   * @return        : 无
13   */
14  void lcd_init(void)
15  {
16      lcdgpio_init();             /* 初始化 IO        */
17      lcdclk_init(32, 3, 5);      /* 初始化 LCD 时钟  */
18  
19      lcd_reset();                /* 复位 LCD         */
20      delayms(10);                /* 延时 10ms        */
21      lcd_noreset();              /* 结束复位         */
22  
23      /* RGB LCD 参数结构体初始化 */
24      tftlcd_dev.height = 600;    /* 屏幕高度         */
25      tftlcd_dev.width = 1024;    /* 屏幕宽度         */
```

```
26      tftlcd_dev.pixsize = 4;       /* ARGB8888 模式，每个像素 4 字节   */
27      tftlcd_dev.vspw = 3;          /* VSYNC 信号宽度                   */
28      tftlcd_dev.vbpd = 20;         /* 帧同步信号后肩                   */
29      tftlcd_dev.vfpd = 12;         /* 帧同步信号前肩                   */
30      tftlcd_dev.hspw = 20;         /* HSYNC 信号宽度                   */
31      tftlcd_dev.hbpd = 140;        /* 水平同步信号后见肩               */
32      tftlcd_dev.hfpd = 160;        /* 水平同步信号前肩                 */
33      tftlcd_dev.framebuffer = LCD_FRAMEBUF_ADDR; /* 帧缓冲地址         */
34      tftlcd_dev.backcolor = LCD_WHITE;   /* 背景色为白色               */
35      tftlcd_dev.forecolor = LCD_BLACK;   /* 前景色为黑色               */
36
37      /* 初始化 ELCDIF 的 CTRL 寄存器
38       * bit [31] 0 : 停止复位
39       * bit [19] 1 : 旁路计数器模式
40       * bit [17] 1 : LCD 工作在 dotclk 模式
41       * bit [15:14] 00 : 输入数据不交换
42       * bit [13:12] 00 : CSC 不交换
43       * bit [11:10] 11 : 24 位总线宽度
44       * bit [9:8]   11 : 24 位数据宽度,也就是 RGB888
45       * bit [5]     1  : elcdif 工作在主模式
46       * bit [1]     0  : 所有的 24 位均有效
47       */
48       LCDIF->CTRL |= (1 << 19) | (1 << 17) | (0 << 14) | (0 << 12) |
49                      (3 << 10) | (3 << 8) | (1 << 5) | (0 << 1);
50      /*
51       * 初始化 ELCDIF 的寄存器 CTRL1
52       * bit [19:16]  : 0X7 ARGB 模式下，传输 24 位数据，A 通道不用传输
53       */
54       LCDIF->CTRL1 = 0X7 << 16;
55
56       /*
57        * 初始化 ELCDIF 的寄存器 TRANSFER_COUNT 寄存器
58        * bit [31:16]  : 高度
59        * bit [15:0]   : 宽度
60        */
61      LCDIF->TRANSFER_COUNT  = (tftlcd_dev.height << 16) |
                                 (tftlcd_dev.width << 0);
62
63      /*
64       * 初始化 ELCDIF 的 VDCTRL0 寄存器
65       * bit [29] 0 : VSYNC 输出
66       * bit [28] 1 : 使能 ENABLE 输出
67       * bit [27] 0 : VSYNC 低电平有效
```

```
68         * bit [26] 0 : HSYNC 低电平有效
69         * bit [25] 0 : DOTCLK 上升沿有效
70         * bit [24] 1 : ENABLE 信号高电平有效
71         * bit [21] 1 : DOTCLK 模式下设置为 1
72         * bit [20] 1 : DOTCLK 模式下设置为 1
73         * bit [17:0] : vspw 参数
74         */
75        LCDIF->VDCTRL0 = 0; /* 先清零 */
76        LCDIF->VDCTRL0 = (0 << 29) | (1 << 28) | (0 << 27) |
77                         (0 << 26) | (0 << 25) | (1 << 24) |
78                         (1 << 21) | (1 << 20) | (tftlcd_dev.vspw << 0);
79        /*
80         * 初始化 ELCDIF 的 VDCTRL1 寄存器，设置 VSYNC 总周期
81         */
82        LCDIF->VDCTRL1 = tftlcd_dev.height + tftlcd_dev.vspw +
                          tftlcd_dev.vfpd + tftlcd_dev.vbpd;
83
84         /*
85          * 初始化 ELCDIF 的 VDCTRL2 寄存器，设置 HSYNC 周期
86          * bit[31:18] : hsw
87          * bit[17:0]  : HSYNC 总周期
88          */
89        LCDIF->VDCTRL2 = (tftlcd_dev.hspw << 18) | (tftlcd_dev.width +
                    tftlcd_dev.hspw + tftlcd_dev.hfpd + tftlcd_dev.hbpd);
90
91        /*
92         * 初始化 ELCDIF 的 VDCTRL3 寄存器，设置 HSYNC 周期
93         * bit[27:16] : 水平等待时钟数
94         * bit[15:0]  : 垂直等待时钟数
95         */
96        LCDIF->VDCTRL3 = ((tftlcd_dev.hbpd + tftlcd_dev.hspw) << 16) |
                           (tftlcd_dev.vbpd + tftlcd_dev.vspw);
97
98        /*
99         * 初始化 ELCDIF 的 VDCTRL4 寄存器，设置 HSYNC 周期
100        * bit[18] 1 : 当使用 VSHYNC、HSYNC、DOTCLK 的话此为置 1
101        * bit[17:0]  : 宽度
102        */
103
104       LCDIF->VDCTRL4 = (1<<18) | (tftlcd_dev.width);
105
106       /*
107        * 初始化 ELCDIF 的 CUR_BUF 和 NEXT_BUF 寄存器
```

```
108      * 设置当前显存地址和下一帧的显存地址
109      */
110     LCDIF->CUR_BUF = (unsigned int)tftlcd_dev.framebuffer;
111     LCDIF->NEXT_BUF = (unsigned int)tftlcd_dev.framebuffer;
112 
113     lcd_enable();            /* 使能 LCD     */
114     delayms(10);
115     lcd_clear(LCD_WHITE);    /* 清屏         */
116 
117 }
118 
119 /*
120  * @description  : LCD GPIO 初始化
121  * @param        : 无
122  * @return       : 无
123  */
124 void lcdgpio_init(void)
125 {
126     gpio_pin_config_t gpio_config;
127 
128     /* 1、IO 初始化复用功能 */
129     IOMUXC_SetPinMux(IOMUXC_LCD_DATA00_LCDIF_DATA00,0);
130     IOMUXC_SetPinMux(IOMUXC_LCD_DATA01_LCDIF_DATA01,0);
131     IOMUXC_SetPinMux(IOMUXC_LCD_DATA02_LCDIF_DATA02,0);
132     IOMUXC_SetPinMux(IOMUXC_LCD_DATA03_LCDIF_DATA03,0);
......
154     IOMUXC_SetPinMux(IOMUXC_LCD_ENABLE_LCDIF_ENABLE,0);
155     IOMUXC_SetPinMux(IOMUXC_LCD_HSYNC_LCDIF_HSYNC,0);
156     IOMUXC_SetPinMux(IOMUXC_LCD_VSYNC_LCDIF_VSYNC,0);
157     IOMUXC_SetPinMux(IOMUXC_GPIO1_IO08_GPIO1_IO08,0); /* 背光引脚*/
158 
159     /* 2、配置 LCD IO 属性
160      *bit 16:0 HYS 关闭
161      *bit [15:14]: 0 默认 22K 上拉
162      *bit [13]: 0 pull 功能
163      *bit [12]: 0 pull/keeper 使能
164      *bit [11]: 0 关闭开路输出
165      *bit [7:6]: 10 速度 100Mhz
166      *bit [5:3]: 111 驱动能力为 R0/7
167      *bit [0]: 1 高转换率
168      */
169     IOMUXC_SetPinConfig(IOMUXC_LCD_DATA00_LCDIF_DATA00,0xB9);
170     IOMUXC_SetPinConfig(IOMUXC_LCD_DATA01_LCDIF_DATA01,0xB9);
```

```
......
193     IOMUXC_SetPinConfig(IOMUXC_LCD_CLK_LCDIF_CLK,0xB9);
194     IOMUXC_SetPinConfig(IOMUXC_LCD_ENABLE_LCDIF_ENABLE,0xB9);
195     IOMUXC_SetPinConfig(IOMUXC_LCD_HSYNC_LCDIF_HSYNC,0xB9);
196     IOMUXC_SetPinConfig(IOMUXC_LCD_VSYNC_LCDIF_VSYNC,0xB9);
197     IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO08_GPIO1_IO08,0xB9);
198 
199     /* GPIO 初始化 */
200     gpio_config.direction = kGPIO_DigitalOutput; /* 输出         */
201     gpio_config.outputLogic = 1;                 /* 默认关闭背光 */
202     gpio_init(GPIO1, 8, &gpio_config);           /* 背光默认打开 */
203     gpio_pinwrite(GPIO1, 8, 1);                  /* 打开背光     */
204 }
205 
206 /*
207  * @description      : LCD 时钟初始化, LCD 时钟计算公式如下:
208  *                      LCD CLK = 24 * loopDiv / prediv / div
209  * @param - loopDiv  : loopDivider 值
210  * @param - loopDiv  : lcdifprediv 值
211  * @param - div      : lcdifdiv 值
212  * @return           : 无
213  */
214 void lcdclk_init(unsigned char loopDiv, unsigned char prediv,
                    unsigned char div)
215 {
216     /* 先初始化 video pll
217      * VIDEO PLL = OSC24M * (loopDivider + (denominator /
                   numerator)) / postDivider
218      *不使用小数分频器，因此 denominator 和 numerator 设置为 0
219      */
220     CCM_ANALOG->PLL_VIDEO_NUM = 0;      /* 不使用小数分频器 */
221     CCM_ANALOG->PLL_VIDEO_DENOM = 0;
222 
223     /*
224      * PLL_VIDEO 寄存器设置
225      * bit[13]    : 1   使能 VIDEO PLL 时钟
226      * bit[20:19] : 2   设置 postDivider 为 1 分频
227      * bit[6:0]   : 32  设置 loopDivider 寄存器
228      */
229     CCM_ANALOG->PLL_VIDEO =  (2 << 19) | (1 << 13) | (loopDiv << 0);
230 
231     /*
232      * MISC2 寄存器设置
```

```
233      * bit[31:30]: 0  VIDEO 的 post-div 设置，1 分频
234      */
235     CCM_ANALOG->MISC2 &= ~(3 << 30);
236     CCM_ANALOG->MISC2 = 0 << 30;
237 
238     /* LCD 时钟源来源与 PLL5，也就是 VIDEO PLL  */
239     CCM->CSCDR2 &= ~(7 << 15);
240     CCM->CSCDR2 |= (2 << 15);     /* 设置 LCDIF_PRE_CLK 使用 PLL5 */
241 
242     /* 设置 LCDIF_PRE 分频 */
243     CCM->CSCDR2 &= ~(7 << 12);
244     CCM->CSCDR2 |= (prediv - 1) << 12;     /* 设置分频  */
245 
246     /* 设置 LCDIF 分频 */
247     CCM->CBCMR &= ~(7 << 23);
248     CCM->CBCMR |= (div - 1) << 23;
249 
250     /* 设置 LCD 时钟源为 LCDIF_PRE 时钟 */
251     CCM->CSCDR2 &= ~(7 << 9);   /* 清除原来的设置            */
252     CCM->CSCDR2 |= (0 << 9);    /* LCDIF_PRE 时钟源选择 LCDIF_PRE 时钟 */
253 }
254 
255 /*
256  * @description  : 复位 ELCDIF 接口
257  * @param        : 无
258  * @return       : 无
259  */
260 void lcd_reset(void)
261 {
262     LCDIF->CTRL  = 1<<31;    /* 强制复位     */
263 }
264 
265 /*
266  * @description  : 结束复位 ELCDIF 接口
267  * @param        : 无
268  * @return       : 无
269  */
270 void lcd_noreset(void)
271 {
272     LCDIF->CTRL  = 0<<31;    /* 取消强制复位  */
273 }
274 
275 /*
```

```
276  * @description   : 使能 ELCDIF 接口
277  * @param         : 无
278  * @return        : 无
279  */
280 void lcd_enable(void)
281 {
282     LCDIF->CTRL |= 1<<0; /* 使能 ELCDIF */
283 }
284 
285 /*
286  * @description       : 画点函数
287  * @param - x         : x 轴坐标
288  * @param - y         : y 轴坐标
289  * @param - color     : 颜色值
290  * @return            : 无
291  */
292 inline void lcd_drawpoint(unsigned short x,unsigned short y,
                              unsigned int color)
293 {
294     *(unsigned int*)((unsigned int)tftlcd_dev.framebuffer +
295                      tftlcd_dev.pixsize * (tftlcd_dev.width *
                         y + x)) = color;
296 }
297 
298 
299 /*
300  * @description   : 读取指定点的颜色值
301  * @param - x     : x 轴坐标
302  * @param - y     : y 轴坐标
303  * @return        : 读取到的指定点的颜色值
304  */
305 inline unsigned int lcd_readpoint(unsigned short x,
                                      unsigned short y)
306 {
307     return *(unsigned int*)((unsigned int)tftlcd_dev.framebuffer +
308            tftlcd_dev.pixsize * (tftlcd_dev.width * y + x));
309 }
310 
311 /*
312  * @description       : 清屏
313  * @param - color     : 颜色值
314  * @return            : 读取到的指定点的颜色值
315  */
```

```
316 void lcd_clear(unsigned int color)
317 {
318     unsigned int num;
319     unsigned int i = 0;
320
321     unsigned int *startaddr=(unsigned int*)tftlcd_dev.framebuffer;
322     num=(unsigned int)tftlcd_dev.width * tftlcd_dev.height;
323     for(i = 0; i < num; i++)
324     {
325         startaddr[i] = color;
326     }
327 }
328
329 /*
330  * @description       : 以指定的颜色填充一块矩形
331  * @param - x0        : 矩形起始点坐标 X 轴
332  * @param - y0        : 矩形起始点坐标 Y 轴
333  * @param - x1        : 矩形终止点坐标 X 轴
334  * @param - y1        : 矩形终止点坐标 Y 轴
335  * @param - color     : 要填充的颜色
336  * @return            : 读取到的指定点的颜色值
337  */
338 void lcd_fill(unsigned   short x0, unsigned short y0,
339               unsigned short x1, unsigned short y1,
               unsigned int color)
340 {
341     unsigned short x, y;
342
343     if(x0 < 0) x0 = 0;
344     if(y0 < 0) y0 = 0;
345     if(x1 >= tftlcd_dev.width) x1 = tftlcd_dev.width - 1;
346     if(y1 >= tftlcd_dev.height) y1 = tftlcd_dev.height - 1;
347
348     for(y = y0; y <= y1; y++)
349     {
350         for(x = x0; x <= x1; x++)
351             lcd_drawpoint(x, y, color);
352     }
353 }
```

文件 bsp_lcd.c 里面一共有 10 个函数，第一个函数是 lcd_init，这个是 LCD 初始化函数，此函数先调用 LCD 的 IO 初始化函数、时钟初始化函数、复位函数等，然后会按照我们前面讲解的步骤初始化 eLCDIF 相关的寄存器，最后使能 eLCDIF。第二个函数是 lcdgpio_init，这个是 LCD 的 IO 初始化函数。第三个函数 lcdclk_init 是 LCD 的时钟初始化函数。第四个函数 lcd_reset

和第五个函数 lcd_noreset 分别为复位 LCD 的停止 LCD 复位函数。第六个函数 lcd_enable 是 eLCDIF 使能函数，用于使能 eLCDIF。第七个和第八个是画点和读点函数，分别为 lcd_drawpoint 和 lcd_readpoint，通过这两个函数就可以在 LCD 的指定像素点上显示指定的颜色，或者读取指定像素点的颜色。第九个函数 lcd_clear 是清屏函数，使用指定的颜色清除整个屏幕。最后一个函数 lcd_fill 是填充函数，使用此函数的时候需要指定矩形的起始坐标、终止坐标和填充颜色，这样就可以填充出一个矩形区域。

在 bsp_lcdapi.h 中输入如下所示内容：

示例代码 24.3.3 bsp_lcdapi.h 文件代码

```
1  #ifndef BSP_LCDAPI_H
2  #define BSP_LCDAPI_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_lcdapi.h
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : LCD 显示 API 函数。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/3/18 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 #include "bsp_lcd.h"
15 
16 /* 函数声明 */
17 void lcd_drawline(unsigned short x1, unsigned short y1, unsigned
                      short x2, unsigned short y2);
18 void lcd_draw_rectangle(unsigned short x1, unsigned short y1,
                           unsigned short x2, unsigned short y2);
19 void lcd_draw_circle(unsigned short x0,unsigned short y0,
                        unsigned char r);
20 void lcd_showchar(unsigned short x,unsigned short y,
                      unsigned char num,unsigned char size,
                      unsigned char mode);
21 unsigned int lcd_pow(unsigned char m,unsigned char n);
22 void lcd_shownum(unsigned short x, unsigned short y,
                    unsigned int num, unsigned char len,
                    unsigned char size);
23 void lcd_showxnum(unsigned short x, unsigned short y,
                      unsigned int num, unsigned char len,
                      unsigned char size, unsigned char mode);
24 void lcd_show_string(unsigned short x,unsigned short y,
                        unsigned short width, unsigned short height,
                        unsigned char size, char *p);
```

```
25 #endif
```

文件 bsp_lcdapi.h 内容很简单，就是函数声明。在 bsp_lcdapi.c 中输入如下内容：

示例代码 24.3.4 bsp_lcdapi.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : bsp_lcdapi.c
作者     : 左忠凯
版本     : V1.0
描述     : LCD API 函数文件。
其他     : 无
论坛     : www.openedv.com
日志     : 初版 V1.0 2019/3/18 左忠凯创建
***************************************************************/
1  #include "bsp_lcdapi.h"
2  #include "font.h"
3  
4  /*
5   * @description       : 画线函数
6   * @param - x1        : 线起始点坐标 X 轴
7   * @param - y1        : 线起始点坐标 Y 轴
8   * @param - x2        : 线终止点坐标 X 轴
9   * @param - y2        : 线终止点坐标 Y 轴
10  * @return            : 无
11  */
12 void lcd_drawline(unsigned short x1, unsigned short y1,
                     unsigned short x2, unsigned short y2)
13 {
14     u16 t;
15     int xerr = 0, yerr = 0, delta_x, delta_y, distance;
16     int incx, incy, uRow, uCol;
17     delta_x = x2 - x1;                  /* 计算坐标增量    */
18     delta_y = y2 - y1;
19     uRow = x1;
20     uCol = y1;
21     if(delta_x > 0) incx = 1;          /* 设置单步方向     */
22     else if(delta_x==0) incx = 0;      /* 垂直线           */
23     else
24     {
25         incx = -1;
26         delta_x = -delta_x;
27     }
28 
```

```
29      if(delta_y>0) incy=1;
30      else if(delta_y == 0)  incy=0;     /* 水平线          */
31      else
32      {
33          incy = -1;
34          delta_y = -delta_y;
35      }
36      if( delta_x > delta_y)  distance = delta_x; /*选取基本增量坐标轴  */
37      else distance = delta_y;
38      for(t = 0; t <= distance+1; t++ )           /* 画线输出 */
39      {
40          lcd_drawpoint(uRow, uCol, tftlcd_dev.forecolor);/* 画点 */
41          xerr += delta_x ;
42          yerr += delta_y ;
43          if(xerr > distance)
44          {
45              xerr -= distance;
46              uRow += incx;
47          }
48          if(yerr > distance)
49          {
50              yerr -= distance;
51              uCol += incy;
52          }
53      }
54  }
55
56  /*
57   * @description    : 画矩形函数
58   * @param - x1     : 矩形坐上角坐标 X 轴
59   * @param - y1     : 矩形坐上角坐标 Y 轴
60   * @param - x2     : 矩形右下角坐标 X 轴
61   * @param - y2     : 矩形右下角坐标 Y 轴
62   * @return         : 无
63   */
64  void lcd_draw_rectangle(unsigned short x1, unsigned short y1,
                            unsigned short x2, unsigned short y2)
65  {
66      lcd_drawline(x1, y1, x2, y1);
67      lcd_drawline(x1, y1, x1, y2);
68      lcd_drawline(x1, y2, x2, y2);
69      lcd_drawline(x2, y1, x2, y2);
70  }
```

```
71
72 /*
73  * @description   : 在指定位置画一个指定大小的圆
74  * @param - x0    : 圆心坐标 X 轴
75  * @param - y0    : 圆心坐标 Y 轴
76  * @param - y2    : 圆形半径
77  * @return        : 无
78  */
79 void lcd_draw_circle(unsigned short x0,unsigned short y0,
                        unsigned char r)
80 {
81     int mx = x0, my = y0;
82     int x = 0, y = r;
83
84     int d = 1 - r;
85     while(y > x)       /* y>x 即第一象限的第 1 区八分圆 */
86     {
87         lcd_drawpoint(x  + mx, y  + my, tftlcd_dev.forecolor);
88         lcd_drawpoint(y  + mx, x  + my, tftlcd_dev.forecolor);
89         lcd_drawpoint(-x + mx, y  + my, tftlcd_dev.forecolor);
90         lcd_drawpoint(-y + mx, x  + my, tftlcd_dev.forecolor);
91
92         lcd_drawpoint(-x + mx, -y + my, tftlcd_dev.forecolor);
93         lcd_drawpoint(-y + mx, -x + my, tftlcd_dev.forecolor);
94         lcd_drawpoint(x  + mx, -y + my, tftlcd_dev.forecolor);
95         lcd_drawpoint(y  + mx, -x + my, tftlcd_dev.forecolor);
96         if( d < 0)
97         {
98             d = d + 2 * x + 3;
99         }
100        else
101        {
102            d= d + 2 * (x - y) + 5;
103            y--;
104        }
105        x++;
106    }
107 }
108
109 /*
110  * @description   : 在指定位置显示一个字符
111  * @param - x     : 起始坐标 X 轴
112  * @param - y     : 起始坐标 Y 轴
```

```
113  * @param - num   : 显示字符
114  * @param – size  : 字体大小, 可选 12/16/24/32
115  * @param – mode  : 叠加方式(1)还是非叠加方式(0)
116  * @return        : 无
117  */
118 void lcd_showchar(unsigned    short x, unsigned short y,
119                   unsigned char num, unsigned char size,
120                   unsigned char mode)
121 {
122     unsigned char  temp, t1, t;
123     unsigned short y0 = y;
    /* 得到字体一个字符对应点阵集所占的字节数    */
124     unsigned char csize = (size / 8+ ((size % 8) ? 1 : 0)) *
                              (size / 2);
125     num = num - ' ';    /* 得到偏移后的值（ASCII 字库是从空格开始取模，
                              所以-' '就是对应字符的字库）   */
126     for(t = 0; t < csize; t++)
127     {
128         if(size == 12) temp = asc2_1206[num][t];   /* 调用 1206 字体 */
129         else if(size == 16)temp = asc2_1608[num][t];/* 调用 1608 字体 */
130         else if(size == 24)temp = asc2_2412[num][t];/* 调用 2412 字体 */
131         else if(size == 32)temp = asc2_3216[num][t];/* 调用 3216 字体 */
132         else return;                               /* 没有的字库     */
133         for(t1 = 0; t1 < 8; t1++)
134         {
135             if(temp & 0x80)lcd_drawpoint(x, y, tftlcd_dev.forecolor);
136             else if(mode==0)lcd_drawpoint(x, y,
                                     tftlcd_dev.backcolor);
137             temp <<= 1;
138             y++;
139             if(y >= tftlcd_dev.height) return;         /* 超区域了 */
140             if((y - y0) == size)
141             {
142                 y = y0;
143                 x++;
144                 if(x >= tftlcd_dev.width) return;      /* 超区域了 */
145                 break;
146             }
147         }
148     }
149 }
150
151 /*
```

```
152  * @description : 计算 m 的 n 次方
153  * @param - m   : 要计算的值
154  * @param - n   : n 次方
155  * @return      : m^n 次方.
156  */
157 unsigned int lcd_pow(unsigned char m,unsigned char n)
158 {
159     unsigned int result = 1;
160     while(n--) result *= m;
161     return result;
162 }
163 
164 /*
165  * @description  : 显示指定的数字，高位为 0 的话不显示
166  * @param - x    : 起始坐标点 X 轴。
167  * @param - y    : 起始坐标点 Y 轴。
168  * @param - num  : 数值(0~999999999)。
169  * @param - len  : 数字位数。
170  * @param – size : 字体大小
171  * @return       : 无
172  */
173 void lcd_shownum(unsigned    short x,
174                  unsigned short y,
175                  unsigned int num,
176                  unsigned char len,
177                  unsigned char size)
178 {
179     unsigned char  t, temp;
180     unsigned char  enshow = 0;
181     for(t = 0; t < len; t++)
182     {
183         temp = (num / lcd_pow(10, len - t - 1)) % 10;
184         if(enshow == 0 && t < (len - 1))
185         {
186             if(temp == 0)
187             {
188                 lcd_showchar(x + (size / 2) * t, y, ' ', size, 0);
189                 continue;
190             }else enshow = 1;
191         }
192         lcd_showchar(x + (size / 2) * t, y, temp + '0', size, 0);
193     }
194 }
```

```
195
196 /*
197  * @description     : 显示指定的数字，高位为0,还是显示
198  * @param - x       : 起始坐标点 X 轴。
199  * @param - y       : 起始坐标点 Y 轴。
200  * @param - num     : 数值(0~999999999)。
201  * @param - len     : 数字位数。
202  * @param - size    : 字体大小
203  * @param - mode    : [7]:0,不填充;1,填充 0.
204  *                    [6:1]:保留
205  *                    [0]:0,非叠加显示;1,叠加显示.
206  * @return          : 无
207  */
208 void lcd_showxnum(unsigned    short x, unsigned short y,
209                   unsigned int num, unsigned char len,
210                   unsigned char size, unsigned char mode)
211 {
212     unsigned char t, temp;
213     unsigned char enshow = 0;
214     for(t = 0; t < len; t++)
215     {
216         temp = (num / lcd_pow(10, len - t- 1)) % 10;
217         if(enshow == 0 && t < (len - 1))
218         {
219             if(temp == 0)
220             {
221                 if(mode & 0X80) lcd_showchar(x + (size / 2) * t, y, \
                                    '0', size, mode & 0X01);
222                 else  lcd_showchar(x + (size / 2) * t, y , ' ', size,
                                    mode & 0X01);
223                 continue;
224             }else enshow=1;
225 
226         }
227         lcd_showchar( x + (size / 2) * t, y, temp + '0' , size ,
                        mode & 0X01);
228     }
229 }
230 
231 /*
232  * @description     : 显示一串字符串
233  * @param - x       : 起始坐标点 X 轴。
234  * @param - y       : 起始坐标点 Y 轴。
```

```
235  * @param - width     : 字符串显示区域长度
236  * @param - height    : 字符串显示区域高度
237  * @param - size     : 字体大小
238  * @param - p         : 要显示的字符串首地址
239  * @return            : 无
240  */
241 void lcd_show_string(unsigned short x,unsigned short y,
242                       unsigned short width,unsigned short height,
243                       unsigned char size,char *p)
244 {
245    unsigned char x0 = x;
246    width += x;
247    height += y;
248    while((*p <= '~') &&(*p >= ' '))       /* 判断是不是非法字符! */
249    {
250        if(x >= width) {x = x0; y += size;}
251        if(y >= height) break;              /* 退出 */
252        lcd_showchar(x, y, *p , size, 0);
253        x += size / 2;
254        p++;
255    }
256 }
```

文件 bsp_lcdapi.h 里面都是一些 LCD 的 API 操作函数，比如画线、画矩形、画圆、显示数字、显示字符和字符串等函数。这些函数都是从 STM32 例程里面移植过来的，如果学习过 ALIENTEK 的 STM32 教程的话就会很熟悉，都是一些纯软件的东西。

lcd_showchar 函数是字符显示函数，要理解这个函数就得先了解一下字符（ASCII 字符集）在 LCD 上的显示原理。要显示字符，我们先要有字符的点阵数据，ASCII 常用的字符集总共有 95 个，从空格符开始，分别为： !"#$%&'()*+,-0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~.

我们先要得到这个字符集的点阵数据，这里我们介绍一个款很好的字符提取软件：PCtoLCD2002 完美版。该软件可以提供各种字符，包括汉字（字体和大小都可以自己设置）阵提取，且取模方式可以设置好几种，常用的取模方式，该软件都支持。该软件还支持图形模式，也就是用户可以自己定义图片的大小，然后画图，根据所画的图形再生成点阵数据，这功能在制作图标或图片的时候很有用。

该软件的界面如图 24.3.1 所示：

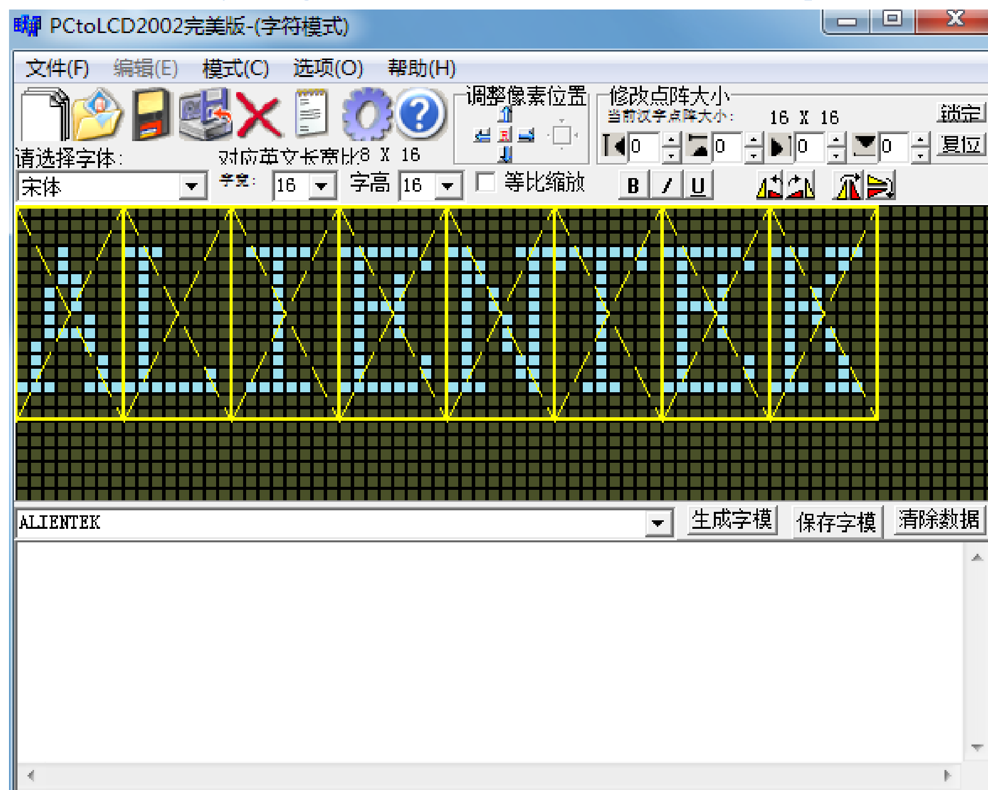


图 24.3.1 PCtoLCD2002 软件界面

然后我们点击字模选项按钮进入字模选项设置界面。设置界面中点阵格式和取模方式等参数配置如图 24.3.2 所示：

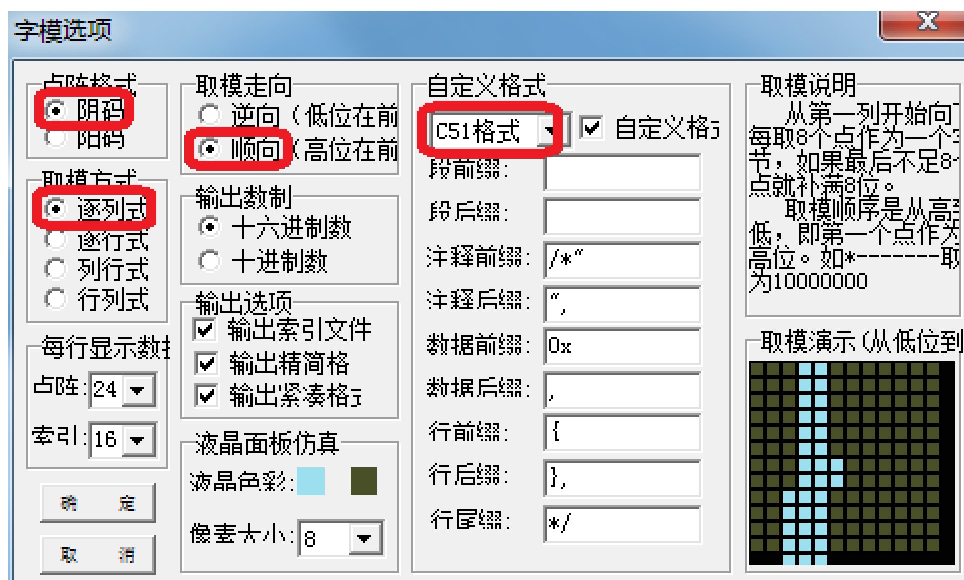


图 24.3.2 设置取模方式

上图设置的取模方式，在右上角的取模说明里面有，即：从第一列开始向下每取 8 个点作为一个字节，如果最后不足 8 个点就补满 8 位。取模顺序是从高到低，即第一个点作为最高位。如*-------取为 10000000。其实就是按如图 24.3.3 所示的这种方式：

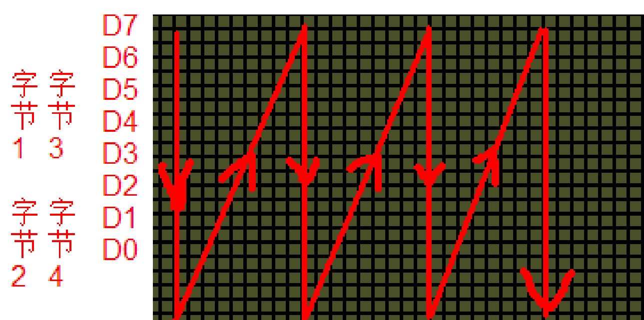


图 24.3.3 取模方式图解

从上到下，从左到右，高位在前。我们按这样的取模方式，然后把 ASCII 字符集按 12*6 大小、16*8、24*12 和 32*16 大小取模出来（对应汉字大小为 12*12、16*16、24*24 和 32*32，字符的只有汉字的一半大!)。将取出的点阵数组保存在 font.h 里面，每个 12*6 的字符占用 12 个字节，每个 16*8 的字符占用 16 个字节，每个 24*12 的字符占用 36 个字节，每个 32*16 的字符占用 64 个字节。font.h 中的字符集点阵数据数组 asc2_1206、asc2_1608、asc2_2412 和 asc2_3216 就对应着这四个大小字符集，具体见 font.h 部分代码（该部分我们不再这里列出来了，请大家参考光盘里面的代码）。

最后在 main.c 中输入如下所示内容：

示例代码 24.3.5 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 16 LCD 液晶屏实验
其他      : 本实验学习如何在 I.MX6U 上驱动 RGB LCD 液晶屏幕，I.MX6U 有个
             ELCDIF 接口，通过此接口可以连接一个 RGB LCD 液晶屏。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "stdio.h"
9  #include "bsp_lcd.h"
10 #include "bsp_lcdapi.h"
11
12
13 /* 背景颜色数组 */
```

```
14 unsigned int backcolor[10] = {
15  LCD_BLUE,       LCD_GREEN,      LCD_RED,    LCD_CYAN,   LCD_YELLOW,
16  LCD_LIGHTBLUE,  LCD_DARKBLUE,   LCD_WHITE,  LCD_BLACK,  LCD_ORANGE
17
18 };
19
20 /*
21  * @description   : main 函数
22  * @param         : 无
23  * @return        : 无
24  */
25 int main(void)
26 {
27     unsigned char index = 0;
28     unsigned char state = OFF;
29
30     int_init();                     /* 初始化中断(一定要最先调用！) */
31     imx6u_clkinit();                /* 初始化系统时钟               */
32     delay_init();                   /* 初始化延时                   */
33     clk_enable();                   /* 使能所有的时钟               */
34     led_init();                     /* 初始化 led                   */
35     beep_init();                    /* 初始化 beep                  */
36     uart_init();                    /* 初始化串口，波特率 115200    */
37     lcd_init();                     /* 初始化 LCD                   */
38
39     tftlcd_dev.forecolor = LCD_RED;
40     lcd_show_string(10,10,400,32,32,(char*)"ALPHA-IMX6UL
                                         ELCD TEST");
41     lcd_draw_rectangle(10, 52, 1014, 290);    /* 绘制矩形框     */
42     lcd_drawline(10, 52,1014, 290);           /* 绘制线条       */
43     lcd_drawline(10, 290,1014, 52);           /* 绘制线条       */
44     lcd_draw_Circle(512, 171, 119);           /* 绘制圆形       */
45
46     while(1)
47     {
48         index++;
49         if(index == 10) index = 0;
50         lcd_fill(0, 300, 1023, 599, backcolor[index]);
51         lcd_show_string(800,10,240,32,32,(char*)"INDEX=");
52         lcd_shownum(896,10, index, 2, 32);  /* 显示数字，叠加显示  */
53
54         state = !state;
55         led_switch(LED0,state);
```

```
56          delayms(1000);                          /* 延时一秒 */
57      }
58      return 0;
59 }
```

第 37 行调用函数 lcd_init 初始化 LCD。

第 39 行设置前景色，也就是画笔颜色为红色。

第 40~44 行都是调用 bsp_lcdapi.c 中的 API 函数在 LCD 上绘制各种图形和显示字符串。

第 46 行的 while 循环中每隔 1S 中就调用函数 lcd_fill 填充指定的区域，并且显示 index 值。

main 函数很简单，重点就是初始化 LCD，然后调用 LCD 的 API 函数进行一些常用的操作，比如画线、画矩形、显示字符串和数字等等。

24.4 编译下载验证

24.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 lcd，然后在在 INCDIRS 和 SRCDIRS 中加入“bsp/lcd”，修改后的 Makefile 如下：

示例代码 19.4.1 Makefile 代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= lcd
3
4   /* 省略掉其它代码...... */
5
6  INCDIRS        :=  imx6ul \
7                     stdio/include \
8                     bsp/clk \
9                     bsp/led \
10                    bsp/delay  \
11                    bsp/beep \
12                    bsp/gpio \
13                    bsp/key \
14                    bsp/exit \
15                    bsp/int \
16                    bsp/epittimer \
17                    bsp/keyfilter \
18                    bsp/uart \
19                    bsp/lcd
20
21 SRCDIRS        :=  project \
22                    stdio/lib \
23                    bsp/clk \
24                    bsp/led \
25                    bsp/delay \
```

```
26                    bsp/beep \
27                    bsp/gpio \
28                    bsp/key \
29                    bsp/exit \
30                    bsp/int \
31                    bsp/epittimer \
32                    bsp/keyfilter \
33                    bsp/uart \
34                    bsp/lcd
35
36 /* 省略掉其它代码...... */
37
38 clean:
39  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“lcd”，也就是目标名称为“lcd”。

第 19 行在变量 INCDIRS 中添加 RGB LCD 驱动头文件(.h)路径。

第 34 行在变量 SRCDIRS 中添加 RGB LCD 驱动驱动文件(.c)路径。

链接脚本保持不变。

24.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 lcd.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload lcd.bin /dev/sdd       //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。程序开始运行，LED0 每隔 1S 闪烁依次，屏幕下半部分会每 1S 刷新依次，并且在屏幕的右上角显示索引值，LCD 屏幕显示如图 24.4.3 所示

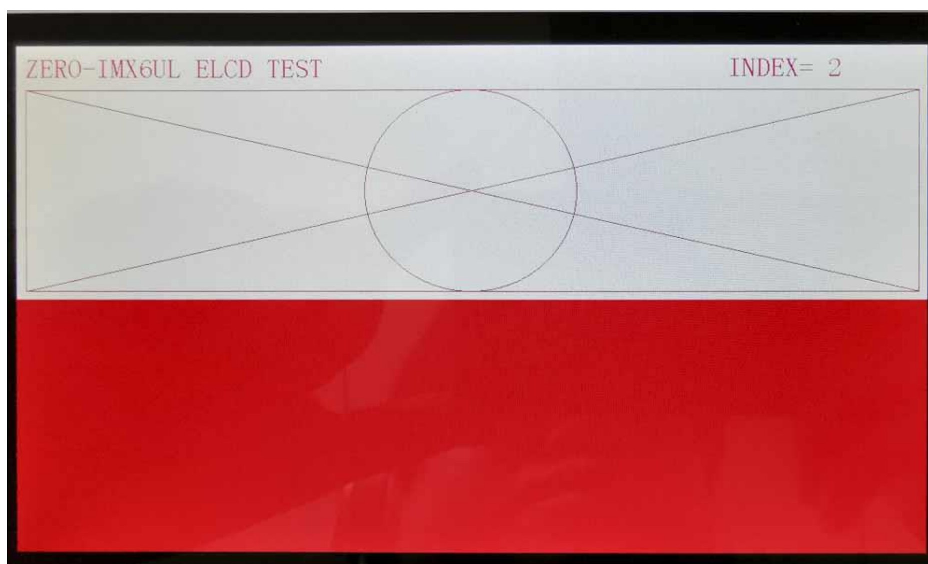


图 24.4.3 LCD 显示画面

第二十五章 RTC 实时时钟实验

实时时钟是很常用的一个外设，通过实时时钟我们就可以知道年、月、日和时间等信息。因此在需要记录时间的场合就需要实时时钟，可以使用专用的实时时钟芯片来完成此功能，但是现在大多数的 MCU 或者 MPU 内部就已经自带了实时时钟外设模块。比如 I.MX6U 内部的 SNVS 就提供了 RTC 功能，本章我们就学习如何使用 I.MX6U 内部的 RTC 来完成实时时钟功能。

25.1 I.MX6U RTC 简介

如果学习过 STM32 的话应该知道，STM32 内部有一个 RTC 外设模块，这个模块需要一个 32.768KHz 的晶振，对这个 RTC 模块进行初始化就可以得到一个实时时钟。I.MX6U 内部也有个 RTC 模块，但是不叫作“RTC”，而是叫做“SNVS”，这一点要注意！本章我们参考《I.MX6UL 参考手册》，而不是《I.MX6ULL 参考手册》，因为《I.MX6ULL 参考手册》很多 SNVS 相关的寄存器并没有给出来，不知道是为何？但是《I.MX6UL 参考手册》里面是完整的。所以本章我们使用《I.MX6UL 参考手册》，如果直接在《I.MX6UL 参考手册》的书签里面找“RTC”相关的字眼是找不到的。I.MX6U 系列的 RTC 是在 SNVS 里面，也就是《I.MX6UL 参考手册》的第 46 章“Chapter 46 Secure Non-Volatile Storage(SNVS)”。

SNVS 直译过来就是安全的非易性存储，SNVS 里面主要是一些低功耗的外设，包括一个安全的实时计数器(RTC)、一个单调计数器(monotonic counter)和一些通用的寄存器，本章我们肯定只使用实时计数器(RTC)。SNVS 里面的外设在芯片掉电以后由电池供电继续运行，I.MX6U-ALPHA 开发板上有一个纽扣电池，这个纽扣电池就是在主电源关闭以后为 SNVS 供电的，如图 25.1.1 所示：

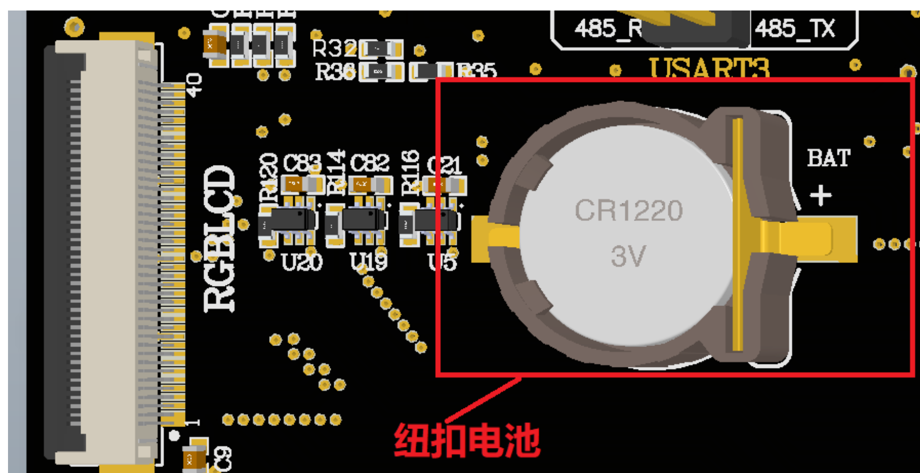


图 25.1.1 I.MX6U-ALPHA 开发板纽扣电池

因为纽扣电池在掉电以后会继续给 SNVS 供电，因此实时计数器就会一直运行，这样的话时间信息就不会丢失，除非纽扣电池没电了。在有纽扣电池作为后备电源的情况下，不管系统主电源是否断电，SNVS 都正常运行。SNVS 有两部分：SNVS_HP 和 SNVS_LP，系统主电源断电以后 SNVS_HP 也会断电，但是在后备电源支持下，SNVS_LP 是不会断电的，而且 SNVS_LP 是和芯片复位隔离开的，因此 SNVS_LP 相关的寄存器的值会一直保存着。

SNVS 分为两个子模块：SNVS_HP 和 SNVS_LP，也就是高功耗域(SNVS_HP)和低功耗域(SNVS_LP)，这两个域的电源来源如下：

SNVS_LP：专用的 always-powered-on 电源域，系统主电源和备用电源都可以为其供电。

SNVS_HP：系统(芯片)电源。

SNVS 的这两个子模块的电源如图 25.1.2 所示：

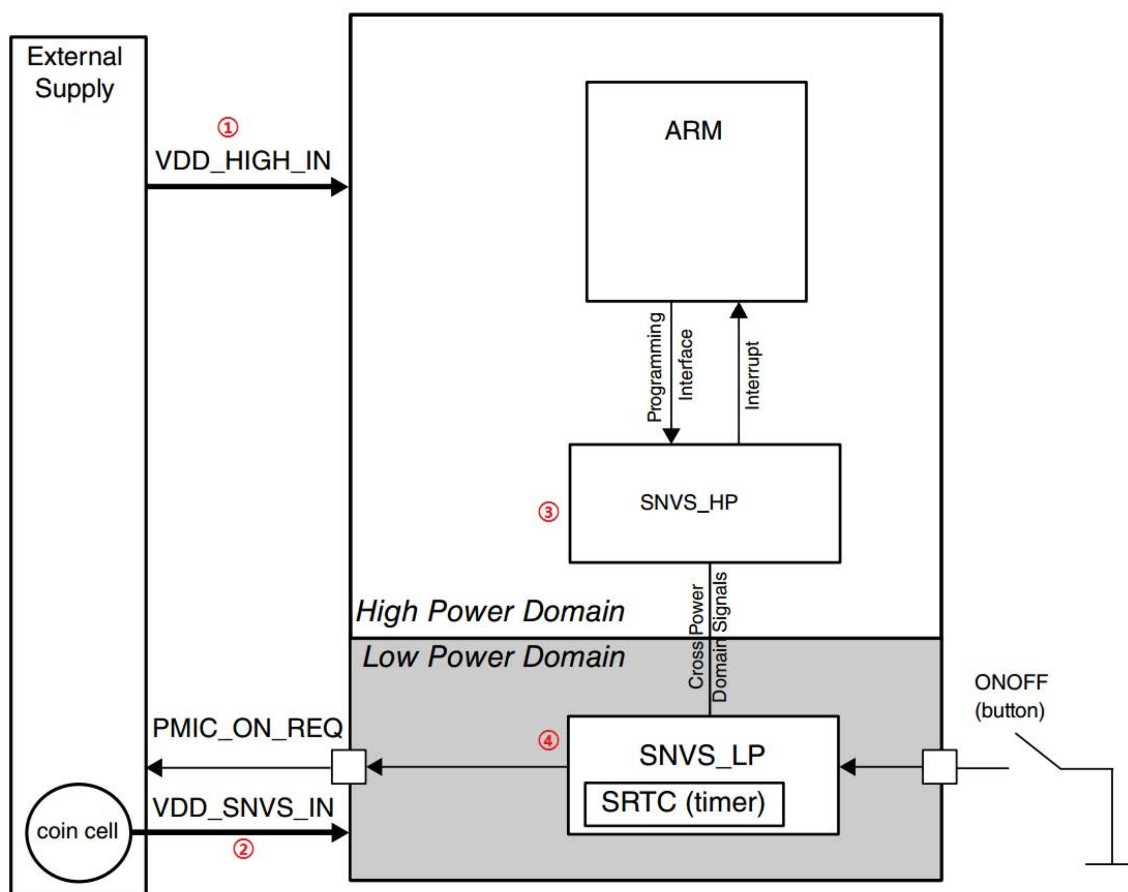


图 25.1.2 SNVS 子模块电源结构图

图 25.1.2 中各个部分功能如下：

、VDD_HIGH_IN 是系统(芯片)主电源，这个电源会同时供给给 SNVS_HP 和 SNVS_LP。

、VDD_SNVS_IN 是纽扣电池供电的电源，这个电源只会供给给 SNVS_LP，保证在系统主电源 VDD_HIGH_IN 掉电以后 SNVS_LP 会继续运行。

、SNVS_HP 部分。

、SNVS_LP 部分，此部分有个 SRTC，这个就是我们本章要使用的 RTC。

其实不管是 SNVS_HP 还是 SNVS_LP，其内部都有一个 SRTC，但是因为 SNVS_HP 在系统电源掉电以后就会关闭，所以我们本章使用的是 SNVS_LP 内部的 SRTC。毕竟我们肯定都不想开发板或者设备每次关闭以后时钟都被清零，然后开机以后先设置时钟。

其实不管是 SNVS_HP 里面的 RTC，还是 SNVS_LP 里面的 SRTC，其本质就是一个定时器，和我们在第八章讲的 EPIT 定时器一样，只要给它提供时钟，它就会一直运行。SRTC 需要外界提供一个 32.768KHz 的时钟，I.MX6U-ALPHA 核心板上的 32.768KHz 的晶振就是提供这个时钟的。寄存器 SNVS_LPSRTCMR 和 SNVS_LPSRTCLR 保存着秒数，直接读取这两个寄存器的值就知道过了多长时间了。一般以 1970 年 1 月 1 日为起点，加上经过的秒数即可得到现在的时间和日期，原理还是很简单的。SRTC 也是带有闹钟功能的，可以在寄存器 SNVS_LPAR 中写入闹钟时间值，当时钟值和闹钟值匹配的时候就会产生闹钟中断，要使用时钟功能的话还需要进行一些设置，本章我们就不使用闹钟了。

接下来我们看一下本章要用到的与 SRTC 相关的部分寄存器，首先是 SNVS_HPCOMR 寄存器，这个寄存器我们只用到了位：NPSWA_EN(bit31)，这个位是非特权软件访问控制位，如果非特权软件要访问 SNVS 的话此位必须为 1。

接下来看一下寄存器SNVS_LPCR寄存器，此寄存器也只用到了一个位：SRTC_ENV(bit0)，此位为 1 的话就使能 STC 计数器。

最后来看一下寄存器 SNVS_SRTCMR 和 SNVS_SRTCLR，这两个寄存器保存着 RTC 的秒数，按照NXP官方的《6UL参考手册》中的说法，SNVS_SRTCMR保存着高15位，SNVS_SRTCLR保存着低 32 位，因此 SRTC 的计数器一共是 47 位。

但是！我在编写驱动的时候发现按照手册上说的去读取计数器值是错误的！具体表现就是时间是混乱的，因此我在查找了 NXP 提供的 SDK 包中的 fsl_snvs_hp.c 以及 Linux 内核中的 rtc-snvs.c 这两个驱动文件以后发现《6UL 参考手册》上对 SNVS_SRTCMR 和 SNVS_SRTCLR 的解释是错误的，经过查阅这两个文件，得到如下结论：

①、SRTC 计数器是 32 位的，不是 47 位！

②、SNVS_SRTCMR 的 bit14:0 这 15 位是 SRTC 计数器的高 15 位。

③、SNVS_SRTCLR 的 bit31:bit15 这 17 位是 SRTC 计数器的低 17 位。

按照上面的解释去读取这两个寄存器就可以得到正确的时间，如果要调整时间的话也是向这两个寄存器写入要设置的时间值对应的秒数就可以了，但是要修改这两个寄存器的话要先关闭 SRTC。

关于 SNVS 中和 RTC 有关的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6UL 参考手册》第 2931 页的 46.7 小节。本章我们使用 I.MX6U 的 SNVS_LP 的 SRTC，配置步骤如下：

1、初始化 SNVS_SRTC

初始化 SNVS_LP 中的 SRTC。

2、设置 RTC 时间

第一次使用 RTC 肯定要先设置时间。

3、使能 RTC

配置好 RTC 并设置好初始时间以后就可以开启 RTC 了。

25.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、RGB LCD 接口。

③、SRTC。

SRTC 需要外接一个 32.768KHz 的晶振，在 I.MX6U-ALPHA 核心板上就有这个 32.768KHz 的晶振，原理图如图 25.2.1 所示：

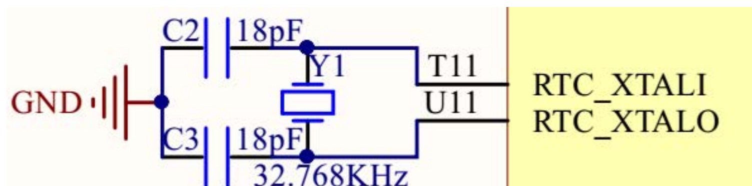


图 25.2.1 外接 32.768KHz 晶振

25.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 16_rtc。

25.3.1 修改文件 MCIMX6Y2.h

在第十三章移植的 NXP 官方 SDK 包是针对 I.MX6ULL 编写的，因此文件 MCIMX6Y2.h 中的结构体 SNVS_Type 里面的寄存器是不全的，我们需要在其中加入本章实验所需要的寄存器，修改 SNVS_Type 为如下所示：

示例代码 25.3.1.1 SNVS_Type 结构体

```
1  typedef struct {
2    __IO uint32_t HPLR;
3    __IO uint32_t HPCOMR;
4    __IO uint32_t HPCR;
5    __IO uint32_t HPSICR;
6    __IO uint32_t HPSVCR;
7    __IO uint32_t HPSR;
8    __IO uint32_t HPSVSR;
9    __IO uint32_t HPHACIVR;
10   __IO uint32_t HPHACR;
11   __IO uint32_t HPRTCMR;
12   __IO uint32_t HPRTCLR;
13   __IO uint32_t HPTAMR;
14   __IO uint32_t HPTALR;
15   __IO uint32_t LPLR;
16   __IO uint32_t LPCR;
17   __IO uint32_t LPMKCR;
18   __IO uint32_t LPSVCR;
19   __IO uint32_t LPTGFCR;
20   __IO uint32_t LPTDCR;
21   __IO uint32_t LPSR;
22   __IO uint32_t LPSRTCMR;
23   __IO uint32_t LPSRTCLR;
24   __IO uint32_t LPTAR;
25   __IO uint32_t LPSMCMR;
26   __IO uint32_t LPSMCLR;
27 }SNVS_Type;
```

25.3.2 编写实验程序

本章实验在上一章例程的基础上完成，更改工程名字为“rtc”，然后在 bsp 文件夹下创建名为“rtc”的文件夹，然后在 bsp/rtc 中新建 bsp_rtc.c 和 bsp_rtc.h 这两个文件。在 bsp_rtc.h 中输入如下内容：

示例代码 25.3.2.1 bsp_rtc.h 文件代码

```
1  #ifndef _BSP_RTC_H
2  #define _BSP_RTC_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
```

```
5  文件名     : bsp_rtc.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : RTC 驱动头文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/1/3 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 相关宏定义 */
16 #define SECONDS_IN_A_DAY     (86400)    /* 一天 86400 秒    */
17 #define SECONDS_IN_A_HOUR    (3600)     /* 一个小时 3600 秒  */
18 #define SECONDS_IN_A_MINUTE  (60)       /* 一分钟 60 秒     */
19 #define DAYS_IN_A_YEAR       (365)      /* 一年 365 天      */
20 #define YEAR_RANGE_START     (1970)     /* 开始年份 1970 年  */
21 #define YEAR_RANGE_END       (2099)     /* 结束年份 2099 年  */
22 
23 /* 时间日期结构体 */
24 struct rtc_datetime
25 {
26     unsigned short year;      /* 范围为:1970 ~ 2099         */
27     unsigned char month;      /* 范围为:1 ~ 12               */
28     unsigned char day;        /* 范围为:1 ~ 31 (不同的月，天数不同).*/
29     unsigned char hour;       /* 范围为:0 ~ 23              */
30     unsigned char minute;     /* 范围为:0 ~ 59              */
31     unsigned char second;     /* 范围为:0 ~ 59              */
32 };
33 
34 /* 函数声明 */
35 void rtc_init(void);
36 void rtc_enable(void);
37 void rtc_disable(void);
38 unsigned int rtc_coverdate_to_seconds(struct rtc_datetime
                                                 *datetime);
39 unsigned int rtc_getseconds(void);
40 void rtc_setdatetime(struct rtc_datetime *datetime);
41 void rtc_getdatetime(struct rtc_datetime *datetime);
42 
43 #endif
```

第 16 到 21 行定义了一些宏，比如一天多少秒、一小时多少秒等等，这些宏将用于将秒转换为时间，或者将时间转换为秒。第 24 行定义了一个结构体 rtc_datetime，此结构体用于描述日期和时间参数。剩下的就是一些函数声明了，很简单。

在文件 bsp_rtc.c 中输入如下内容：

示例代码 25.3.2.2 bsp_rtc.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名   : bsp_rtc.c
作者     : 左忠凯
版本     : V1.0
描述     : RTC 驱动文件。
其他     : 无
论坛     : www.openedv.com
日志     : 初版 V1.0 2019/1/3 左忠凯创建
***************************************************************/
1  #include "bsp_rtc.h"
2  #include "stdio.h"
3  
4  /*
5   * @description   :初始化 RTC
6   */
7  void rtc_init(void)
8  {
9      /*
10      * 设置 HPCOMR 寄存器
11      * bit[31] 1 : 允许访问 SNVS 寄存器，一定要置 1
12      */
13     SNVS->HPCOMR |= (1 << 31);
14 
15 #if 0
16     struct rtc_datetime rtcdate;
17 
18     rtcdate.year = 2018U;
19     rtcdate.month = 12U;
20     rtcdate.day = 13U;
21     rtcdate.hour = 14U;
22     rtcdate.minute = 52;
23     rtcdate.second = 0;
24     rtc_setDatetime(&rtcdate);  /* 初始化时间和日期  */
25 #endif
26     rtc_enable();               /* 使能 RTC        */
27 }
28 
29 /*
30  * @description   : 开启 RTC
31  */
```

```
32 void rtc_enable(void)
33 {
34     /*
35      * LPCR 寄存器 bit0 置 1，使能 RTC
36      */
37     SNVS->LPCR |= 1 << 0;
38     while(!(SNVS->LPCR & 0X01));    /* 等待使能完成 */
39 
40 }
41 
42 /*
43   * @description  : 关闭 RTC
44  */
45 void rtc_disable(void)
46 {
47     /*
48      * LPCR 寄存器 bit0 置 0，关闭 RTC
49      */
50     SNVS->LPCR &= ~(1 << 0);
51     while(SNVS->LPCR & 0X01);   /* 等待关闭完成*/
52 }
53 
54 /*
55  * @description   : 判断指定年份是否为闰年，闰年条件如下：
56  * @param - year  : 要判断的年份
57  * @return        : 1 是闰年，0 不是闰年
58  */
59 unsigned char rtc_isleapyear(unsigned short year)
60 {
61     unsigned char value=0;
62 
63     if(year % 400 == 0)
64         value = 1;
65     else
66     {
67         if((year % 4 == 0) && (year % 100 != 0))
68             value = 1;
69         else
70             value = 0;
71     }
72     return value;
73 }
74 
```

```
75  /*
76   * @description       : 将时间转换为秒数
77   * @param - datetime  : 要转换日期和时间。
78   * @return            : 转换后的秒数
79   */
80  unsigned int rtc_coverdate_to_seconds(struct rtc_datetime *datetime)
81  {
82      unsigned short i = 0;
83      unsigned int seconds = 0;
84      unsigned int days = 0;
85      unsigned short monthdays[] = {0U, 0U, 31U, 59U, 90U, 120U, 151U,
                                     181U, 212U, 243U, 273U, 304U, 334U};
86  
87      for(i = 1970; i < datetime->year; i++)
88      {
89          days += DAYS_IN_A_YEAR;              /* 平年，每年 365 天     */
90          if(rtc_isleapyear(i)) days += 1;    /* 闰年多加一天          */
91      }
92  
93      days += monthdays[datetime->month];
94      if(rtc_isleapyear(i) && (datetime->month >= 3)) days += 1;
95  
96      days += datetime->day - 1;
97  
98      seconds = days * SECONDS_IN_A_DAY +
99                  datetime->hour * SECONDS_IN_A_HOUR +
100                 datetime->minute * SECONDS_IN_A_MINUTE +
101                 datetime->second;
102 
103     return seconds;
104 }
105 
106 /*
107  * @description       : 设置时间和日期
108  * @param - datetime  : 要设置的日期和时间
109  * @return            : 无
110  */
111 void rtc_setdatetime(struct rtc_datetime *datetime)
112 {
113 
114     unsigned int seconds = 0;
115     unsigned int tmp = SNVS->LPCR;
116 
```

```
117     rtc_disable();    /* 设置寄存器 HPRTCMR 和 HPRTCLR 前要先关闭 RTC */
118     /* 先将时间转换为秒         */
119     seconds = rtc_coverdate_to_seconds(datetime);
120     SNVS->LPSRTCMR = (unsigned int)(seconds >> 17); /* 设置高 16 位 */
121     SNVS->LPSRTCLR = (unsigned int)(seconds << 15); /* 设置地 16 位 */
122 
123     /* 如果此前 RTC 是打开的在设置完 RTC 时间以后需要重新打开 RTC */
124     if (tmp & 0x1)
125         rtc_enable();
126 }
127 
128 /*
129  * @description      : 将秒数转换为时间
130  * @param - seconds  : 要转换的秒数
131  * @param – datetime : 转换后的日期和时间
132  * @return           : 无
133  */
134 void rtc_convertseconds_to_datetime(unsigned int seconds,
                                         struct rtc_datetime *datetime)
135 {
136     unsigned int x;
137     unsigned int  secondsRemaining, days;
138     unsigned short daysInYear;
139 
140     /* 每个月的天数       */
141     unsigned char daysPerMonth[] = {0U, 31U, 28U, 31U, 30U, 31U,
                                    30U, 31U, 31U, 30U, 31U, 30U, 31U};
142 
143     secondsRemaining = seconds; /* 剩余秒数初始化 */
144     days = secondsRemaining / SECONDS_IN_A_DAY + 1;
145     secondsRemaining = secondsRemaining % SECONDS_IN_A_DAY;
146 
147     /* 计算时、分、秒 */
148     datetime->hour = secondsRemaining / SECONDS_IN_A_HOUR;
149     secondsRemaining = secondsRemaining % SECONDS_IN_A_HOUR;
150     datetime->minute = secondsRemaining / 60;
151     datetime->second = secondsRemaining % SECONDS_IN_A_MINUTE;
152 
153     /* 计算年 */
154     daysInYear = DAYS_IN_A_YEAR;
155     datetime->year = YEAR_RANGE_START;
156     while(days > daysInYear)
157     {
```

```
158         /* 根据天数计算年 */
159         days -= daysInYear;
160         datetime->year++;
161 
162         /* 处理闰年 */
163         if (!rtc_isleapyear(datetime->year))
164             daysInYear = DAYS_IN_A_YEAR;
165         else    /*闰年，天数加一 */
166             daysInYear = DAYS_IN_A_YEAR + 1;
167     }
168     /*根据剩余的天数计算月份 */
169     if(rtc_isleapyear(datetime->year)) /* 如果是闰年的话 2 月加一天 */
170         daysPerMonth[2] = 29;
171     for(x = 1; x <= 12; x++)
172     {
173         if (days <= daysPerMonth[x])
174         {
175             datetime->month = x;
176             break;
177         }
178         else
179         {
180             days -= daysPerMonth[x];
181         }
182     }
183     datetime->day = days;
184 }
185 
186 /*
187  * @description   : 获取 RTC 当前秒数
188  * @param         : 无
189  * @return        : 当前秒数
190  */
191 unsigned int rtc_getseconds(void)
192 {
193     unsigned int seconds = 0;
194 
195     seconds = (SNVS->LPSRTCMR << 17) | (SNVS->LPSRTCLR >> 15);
196     return seconds;
197 }
198 
199 /*
200  * @description       : 获取当前时间
```

```
201  * @param - datetime : 获取到的时间，日期等参数
202  * @return           : 无
203  */
204 void rtc_getdatetime(struct rtc_datetime *datetime)
205 {
206     unsigned int seconds = 0;
207     seconds = rtc_getseconds();
208     rtc_convertseconds_to_datetime(seconds, datetime);
209 }
```

文件 bsp_rtc.c 里面一共有 9 个函数，依次来看一下这些函数的意义。函数 rtc_init 明显是初始化 rtc 的，主要是使能 RTC，也可以在 rtc_init 函数里面设置时间。函数 rtc_enable 和 rtc_disable 分别是 RTC 的使能和禁止函数。函数 rtc_isleapyear 用于判断某一年是否为闰年。函数 rtc_coverdate_to_seconds 负责将给定的日期和时间信息转换为对应的秒数。函数 rtc_setdatetime 用于设置时间，也就是设置寄存器 SNVS_LPSRTCMR 和 SNVS_LPSRTCLR。函数 rtc_convertseconds_to_datetime 用于将给定的秒数转换为对应的时间值。函数 rtc_getseconds 获取 SRTC 当前秒数，其实就是读取寄存器 SNVS_LPSRTCMR 和 SNVS_LPSRTCLR，然后将其结合成 47 位的值。最后一个函数 rtc_getdatetime 是获取时间值。

我们在 main 函数里面先初始化 RTC，然后进入 3S 倒计时， 如果这 3S 内按下了 KEY0 按键，那么就设置 SRTC 的日期。如果 3S 倒计时结束以后没有按下 KEY0，也就是没有设置 SRTC 时间的话就进入 while 循环，然后读取 RTC 的时间值并且显示在 LCD 上，在文件 main.c 中输入如下所示内容：

示例代码 25.3.2.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名     : mian.c
作者       : 左忠凯
版本       : V1.0
描述       : I.MX6U 开发板裸机实验 17 RTC 实时时钟实验
其他       : 本实验学习如何编写 I.MX6U 内部的 RTC 驱动，使用内部 RTC 可以实现
             一个实时时钟。
论坛       : www.openedv.com
日志       : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "bsp_lcd.h"
9  #include "bsp_lcdapi.h"
10 #include "bsp_rtc.h"
```

```
11 #include "stdio.h"
12
13 /*
14  * @description    : main 函数
15  * @param          : 无
16  * @return         : 无
17  */
18 int main(void)
19 {
20     unsigned char key = 0;
21     int t = 0;
22     int i = 3;            /* 倒计时 3S */
23     char buf[160];
24     struct rtc_datetime rtcdate;
25     unsigned char state = OFF;
26
27     int_init();               /* 初始化中断(一定要最先调用！) */
28     imx6u_clkinit();          /* 初始化系统时钟              */
29     delay_init();             /* 初始化延时                  */
30     clk_enable();             /* 使能所有的时钟              */
31     led_init();               /* 初始化 led                  */
32     beep_init();              /* 初始化 beep                 */
33     uart_init();              /* 初始化串口，波特率 115200   */
34     lcd_init();               /* 初始化 LCD                  */
35     rtc_init();               /* 初始化 RTC                  */
36
37     tftlcd_dev.forecolor = LCD_RED;
38     lcd_show_string(50, 10, 400, 24, 24,      /* 显示字符串 */
                   (char*)"ALPHA-IMX6UL RTC TEST");
39     tftlcd_dev.forecolor = LCD_BLUE;
40     memset(buf, 0, sizeof(buf));
41
42     while(1)
43     {
44         if(t==100)            /* 1s 时间到了 */
45         {
46             t=0;
47             printf("will be running %d s......\r", i);
48
49             lcd_fill(50, 40,370, 70, tftlcd_dev.backcolor); /* 清屏 */
50             sprintf(buf, "will be running %ds......", i);
51             lcd_show_string(50, 40, 300, 24, 24, buf);
52             i--;
```

```
53              if(i < 0)
54                  break;
55          }
56  
57          key = key_getvalue();
58          if(key == KEY0_VALUE)
59          {
60              rtcdate.year = 2018;
61              rtcdate.month = 1;
62              rtcdate.day = 15;
63              rtcdate.hour = 16;
64              rtcdate.minute = 23;
65              rtcdate.second = 0;
66              rtc_setdatetime(&rtcdate); /* 初始化时间和日期 */
67              printf("\r\n RTC Init finish\r\n");
68              break;
69          }
70  
71          delayms(10);
72          t++;
73      }
74      tftlcd_dev.forecolor = LCD_RED;
75      lcd_fill(50, 40,370, 70, tftlcd_dev.backcolor); /* 清屏 */
76      lcd_show_string(50, 40, 200, 24, 24, (char*)"Current Time:");
77      tftlcd_dev.forecolor = LCD_BLUE;
78  
79      while(1)
80      {
81          rtc_getdatetime(&rtcdate);
82          sprintf(buf,"%d/%d/%d %d:%d:%d",rtcdate.year, rtcdate.month,
             rtcdate.day, rtcdate.hour, rtcdate.minute, rtcdate.second);
83          lcd_fill(50,70, 300,94, tftlcd_dev.backcolor);
84          lcd_show_string(50,70,250,24,24,(char*)buf);  /* 显示字符串 */
85  
86          state = !state;
87          led_switch(LED0,state);
88          delayms(1000);  /* 延时一秒 */
89      }
90      return 0;
91  }
```

第 35 行调用函数 rtc_init 初始化 RTC。

第 42 到 73 行是倒计时 3S，如果在这 3S 内按下了 KEY0 按键就会调用函数 rtc_setdatetime 设置当前的时间。如果 3S 到技术结束以后没有按下 KEY0 那就表示不需要设置时间，跳出循环，执行下面的代码。

第 79 到 89 行就是主循环，此循环每隔 1S 调用函数 rtc_getdatetime 获取一次时间值，并且通过串口打印给 SecureCRT 或者在 LCD 上显示。

25.4 编译下载验证

25.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 rtc，然后在在 INCDIRS 和 SRCDIRS 中加入“bsp/rtc”，修改后的 Makefile 如下：

示例代码 25.4.1 Makefile 代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= rtc
3
4 /* 省略掉其它代码...... */
5
6  INCDIRS         :=  imx6ul \
7                      stdio/include \
8                      bsp/clk \
9                      bsp/led \
10                     bsp/delay  \
11                     bsp/beep \
12                     bsp/gpio \
13                     bsp/key \
14                     bsp/exit \
15                     bsp/int \
16                     bsp/epittimer \
17                     bsp/keyfilter \
18                     bsp/uart \
19                     bsp/lcd \
20                     bsp/rtc
21
22 SRCDIRS         :=  project \
23                     stdio/lib \
24                     bsp/clk \
25                     bsp/led \
26                     bsp/delay \
27                     bsp/beep \
28                     bsp/gpio \
29                     bsp/key \
30                     bsp/exit \
31                     bsp/int \
```

```
32                         bsp/epittimer \
33                         bsp/keyfilter \
34                         bsp/uart \
35                         bsp/lcd \
36                         bsp/rtc
37
38 /* 省略掉其它代码...... */
39
40 clean:
41  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“rtc”，也就是目标名称为“rtc”。

第 20 行在变量 INCDIRS 中添加 RTC 驱动头文件(.h)路径。

第 36 行在变量 SRCDIRS 中添加 RTC 驱动驱动文件(.c)路径。

链接脚本保持不变。

25.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 rtc.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload            //给予 imxdownload 可执行权限，一次即可
./imxdownload rtc.bin /dev/sdd   //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。程序一开始进入 3S 倒计时，如图 25.4.2.1 所示：

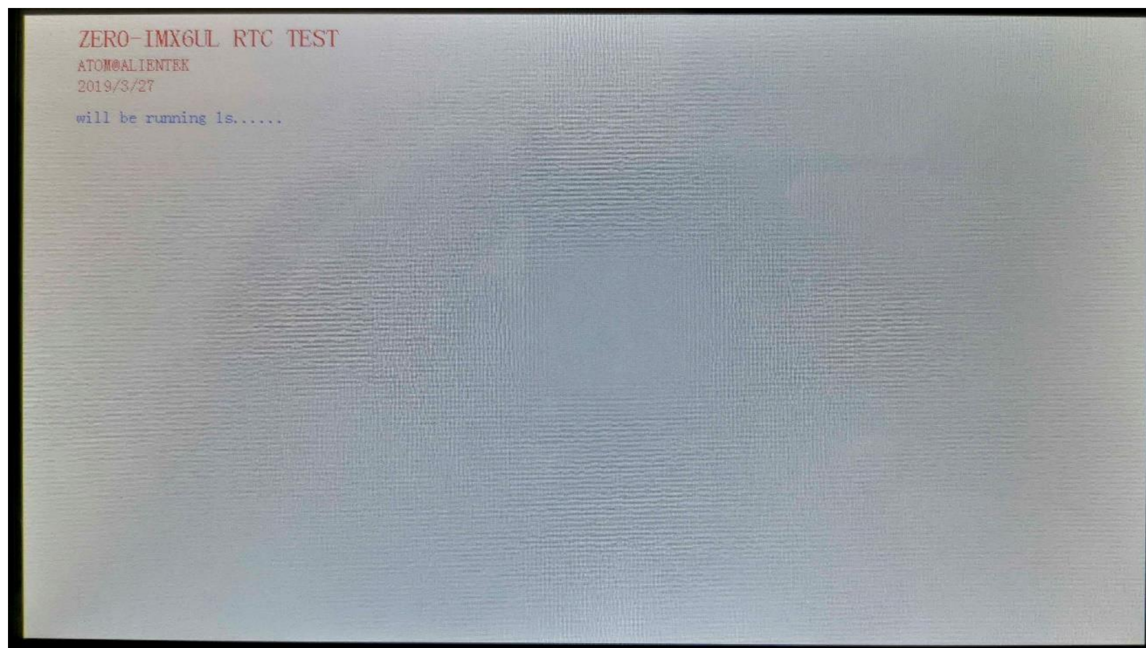


图 24.4.2.1 3 秒钟倒计时

如果在倒计数结束之前按下 KEY0，那么 RTC 就会被设置为我们代码中设置的时间和日期值，RTC 运行如图 24.4.2.2 所示：

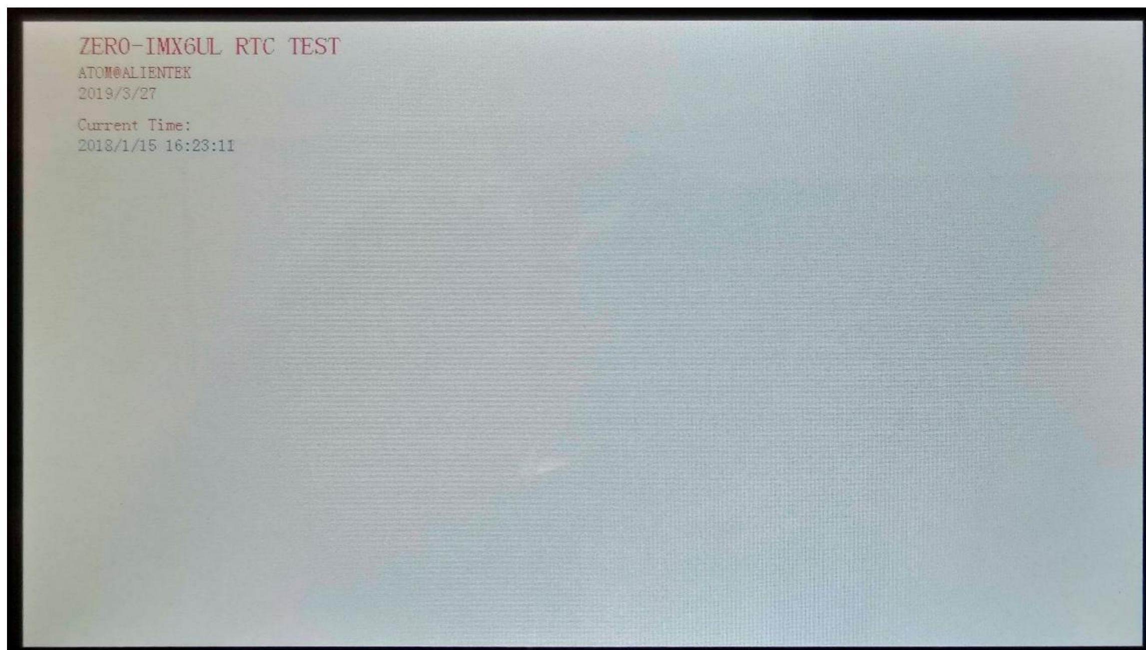


图 24.4.2.2 设置有的时间

我们在 main 函数中设置的时间是 2018 年 1 月 15 日，16 点 23 分 0 秒，在倒计数结束之前按下 KEY0 按键设置 RTC，图 24.4.2.2 中的时间就是我们设置以后的时间。

第二十六章 I2C 实验

I2C 是最常用的通信接口，众多的传感器都会提供 I2C 接口来和主控相连，比如陀螺仪、加速度计、触摸屏等等。所以 I2C 是做嵌入式开发必须掌握的，I.MX6U 有 4 个 I2C 接口，可以通过这 4 个 I2C 接口来连接一些 I2C 外设。I.MX6U-ALPHA 使用 I2C1 接口连接了一个距离传感器 AP3216C，本章我们就来学习如何使用 I.MX6U 的 I2C 接口来驱动 AP3216C，读取 AP3216C 的传感器数据。

26.1 I2C & AP3216C 简介

26.1.1 I2C 简介

I2C 是很常见的一种总线协议，I2C 是 NXP 公司设计的，I2C 使用两条线在主控制器和从机之间进行数据通信。一条是 SCL(串行时钟线)，另外一条是 SDA(串行数据线)，这两条数据线需要接上拉电阻，总线空闲的时候 SCL 和 SDA 处于高电平。I2C 总线标准模式下速度可以达到 100Kb/S，快速模式下可以达到 400Kb/S。I2C 总线工作是按照一定的协议来运行的，接下来就看一下 I2C 协议。

I2C 是支持多从机的，也就是一个 I2C 控制器下可以挂多个 I2C 从设备，这些不同的 I2C 从设备有不同的器件地址，这样 I2C 主控制器就可以通过 I2C 设备的器件地址访问指定的 I2C 设备了，一个 I2C 总线连接多个 I2C 设备如图 26.1.1.1 所示：

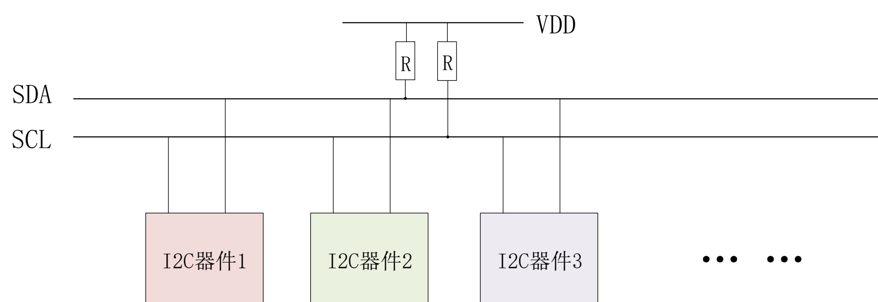


图 26.1.1.1 I2C 多个设备连接结构图

图 26.1.1.1 中 SDA 和 SCL 这两根线必须要接一个上拉电阻，一般是 4.7K。其余的 I2C 从器件都挂接到 SDA 和 SCL 这两根线上，这样就可以通过 SDA 和 SCL 这两根线来访问多个 I2C 设备。

接下来看一下 I2C 协议有关的术语：

1、起始位

顾名思义，也就是 I2C 通信起始标志，通过这个起始位就可以告诉 I2C 从机，“我”要开始进行 I2C 通信了。在 SCL 为高电平的时候，SDA 出现下降沿就表示为起始位，如图 26.1.1.2 所示：

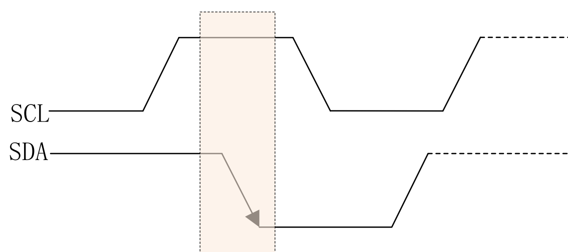


图 26.1.1.2 I2C 通信起始位

2、停止位

停止位就是停止 I2C 通信的标志位，和起始位的功能相反。在 SCL 位高电平的时候，SDA 出现上升沿就表示为停止位，如图 26.1.1.3 所示：

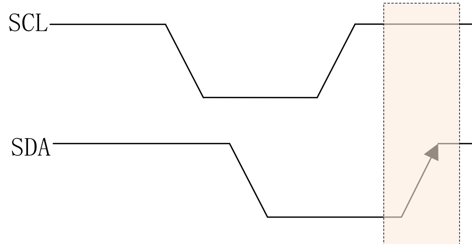


图 26.1.1.3 I2C 通信停止位

3、数据传输

I2C 总线在数据传输的时候要保证在 SCL 高电平期间，SDA 上的数据稳定，因此 SDA 上的数据变化只能在 SCL 低电平期间发生，如图 26.1.1.4 所示：

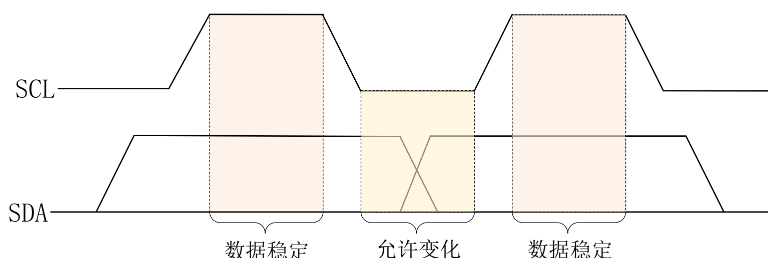


图 26.1.1.4 I2C 数据传输

4、应答信号

当 I2C 主机发送完 8 位数据以后会将 SDA 设置为输入状态，等待 I2C 从机应答，也就是等到 I2C 从机告诉主机它接收到数据了。应答信号是由从机发出的，主机需要提供应答信号所需的时钟，主机发送完 8 位数据以后紧跟着的一个时钟信号就是给应答信号使用的。从机通过将 SDA 拉低来表示发出应答信号，表示通信成功，否则表示通信失败。

5、I2C 写时序

主机通过 I2C 总线与从机之间进行通信不外乎两个操作：写和读，I2C 总线单字节写时序如图 26.1.1.5 所示：

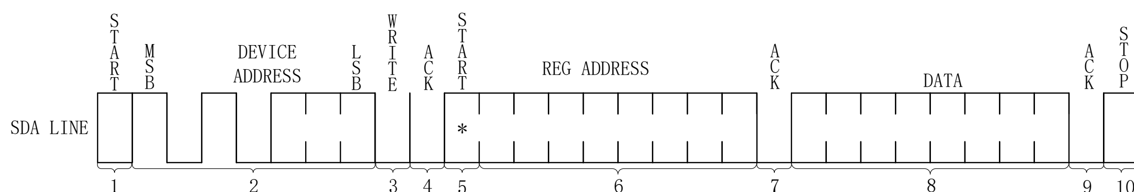


图 26.1.1.5 I2C 写时序

图 26.1.1.5 就是 I2C 写时序，我们来看一下写时序的具体步骤：

1)、开始信号。

2)、发送 I2C 设备地址，每个 I2C 器件都有一个设备地址，通过发送具体的设备地址来决定访问哪个 I2C 器件。这是一个 8 位的数据，其中高 7 位是设备地址，最后 1 位是读写位，为 1 的话表示这是一个读操作，为 0 的话表示这是一个写操作。

3)、 I2C 器件地址后面跟着一个读写位，为 0 表示写操作，为 1 表示读操作。

4)、从机发送的 ACK 应答信号。

5)、重新发送开始信号。

6)、发送要写写入数据的寄存器地址。

7)、从机发送的 ACK 应答信号。

8)、发送要写入寄存器的数据。

9)、从机发送的 ACK 应答信号。

10)、停止信号。

6、I2C 读时序

I2C 总线单字节读时序如图 26.1.1.6 所示：

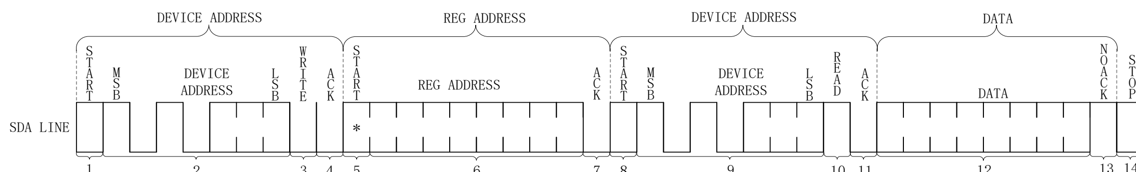


图 26.1.1.6 I2C 单字节读时序

I2C 单字节读时序比写时序要复杂一点，读时序分为 4 大步，第一步是发送设备地址，第二步是发送要读取的寄存器地址，第三步重新发送设备地址，最后一步就是 I2C 从器件输出要读取的寄存器值，我们具体来看一下这步。

1)、主机发送起始信号。

2)、主机发送要读取的 I2C 从设备地址。

3)、读写控制位，因为是向 I2C 从设备发送数据，因此是写信号。

4)、从机发送的 ACK 应答信号。

5)、重新发送 START 信号。

6)、主机发送要读取的寄存器地址。

7)、从机发送的 ACK 应答信号。

8)、重新发送 START 信号。

9)、重新发送要读取的 I2C 从设备地址。

10)、读写控制位，这里是读信号，表示接下来是从 I2C 从设备里面读取数据。

11)、从机发送的 ACK 应答信号。

12)、从 I2C 器件里面读取到的数据。

13)、主机发出 NO ACK 信号，表示读取完成，不需要从机再发送 ACK 信号了。

14)、主机发出 STOP 信号，停止 I2C 通信。

7、I2C 多字节读写时序

又时候我们需要读写多个字节，多字节读写时序和单字节的基本一致，只是在读写数据的时候可以连续发送多个自己的数据，其他的控制时序都是和单字节一样的。

26.1.2 I.MX6U I2C 简介

I.MX6U 提供了 4 个 I2C 外设，通过这四个 I2C 外设即可完成与 I2C 从器件进行通信，I.MX6U 的 I2C 外设特性如下：

①、与标准 I2C 总线兼容。

②、多主机运行

③、软件可编程的 64 中不同的串行时钟序列。

④、软件可选择的应答位。

⑤、开始/结束信号生成和检测。

⑥、重复开始信号生成。

⑦、确认位生成。

⑧、总线忙检测

I.MX6U 的 I2C 支持两种模式：标准模式和快速模式，标准模式下 I2C 数据传输速率最高是 100Kbits/s，在快速模式下数据传输速率最高为 400Kbits/s。

我们接下来看一下 I2C 的几个重要的寄存器，首先看一下 I2Cx_IADR(x=1~4)寄存器，这是 I2C 的地址寄存器，此寄存器结构如图 26.1.2.1 所示：



图 26.1.2.1 寄存器 I2Cx_IADR 结构

寄存器 I2Cx_IADR 只有 ADR(bit7:1)位有效，用来保存 I2C 从设备地址数据。当我们要访问某个 I2C 从设备的时候就需要将其设备地址写入到 ADR 里面。接下来看一下寄存器 I2Cx_IFDR，这个是 I2C 的分频寄存器，寄存器结构如图 26.1.2.2 所示：

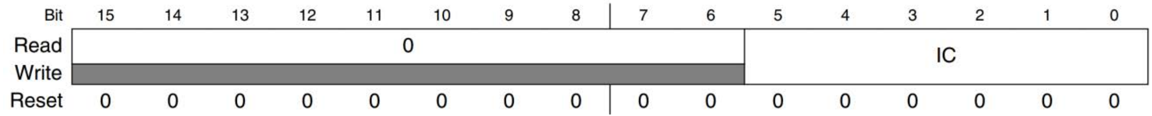


图 26.1.2.2 寄存器 I2Cx_IFDR 结构

寄存器 I2Cx_IFDR 也只有 IC(bit5:0)这个位，用来设置 I2C 的波特率，I2C 的时钟源可以选择 IPG_CLK_ROOT=66MHz，通过设置 IC 位既可以得到想要的 I2C 波特率。IC 位可选的设置如图 26.1.2.3 所示：

IC	Divider	IC	Divider	IC	Divider	IC	Divider
0x00	30	0x10	288	0x20	22	0x30	160
0x01	32	0x11	320	0x21	24	0x31	192
0x02	36	0x12	384	0x22	26	0x32	224
0x03	42	0x13	480	0x23	28	0x33	256
0x04	48	0x14	576	0x24	32	0x34	320
0x05	52	0x15	640	0x25	36	0x35	384
0x06	60	0x16	768	0x26	40	0x36	448
0x07	72	0x17	960	0x27	44	0x37	512
0x08	80	0x18	1152	0x28	48	0x38	640
0x09	88	0x19	1280	0x29	56	0x39	768
0x0A	104	0x1A	1536	0x2A	64	0x3A	896
0x0B	128	0x1B	1920	0x2B	72	0x3B	1024
0x0C	144	0x1C	2304	0x2C	80	0x3C	1280
0x0D	160	0x1D	2560	0x2D	96	0x3D	1536
0x0E	192	0x1E	3072	0x2E	112	0x3E	1792
0x0F	240	0x1F	3840	0x2F	128	0x3F	2048

图 26.1.2.3 IC 设置

不像其他外设的分频设置一样可以随意设置，图 26.1.2.3 中列出了 IC 的所有可选值。比如现在 I2C 的时钟源为 66MHz，我们要设置 I2C 的波特率为 100KHz，那么 IC 就可以设置为 0X15，也就是 640 分频。66000000/640=103.125KHz≈100KHz。

接下来看一下寄存器 I2Cx_I2CR，这个是 I2C 控制寄存器，此寄存器结构如图 26.1.2.4 所示：

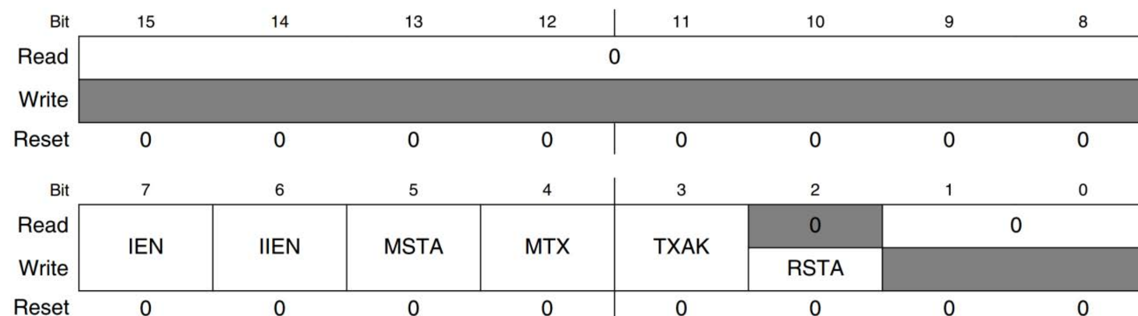


图 26.1.2.4 寄存器 I2Cx_I2CR 结构

寄存器 I2Cx_I2CR 的各位含义如下：

IEN(bit7)：I2C 使能位，为 1 的时候使能 I2C，为 0 的时候关闭 I2C。

IIEN(bit6)：I2C 中断使能位，为 1 的时候使能 I2C 中断，为 0 的时候关闭 I2C 中断。

MSTA(bit5)：主从模式选择位，设置 IIC 工作在主模式还是从模式，为 1 的时候工作在主模式，为 0 的时候工作在从模式。

MTX(bit4)：传输方向选择位，用来设置是进行发送还是接收，为 0 的时候是接收，为 1 的时候是发送。

TXAK(bit3)：传输应答位使能，为 0 的话发送 ACK 信号，为 1 的话发送 NO ACK 信号。

RSTA(bit2)：重复开始信号，为 1 的话产生一个重新开始信号。

接下来看一下寄存器 I2Cx_I2SR，这个是 I2C 的状态寄存器，寄存器结构如图 26.1.2.5 所示：

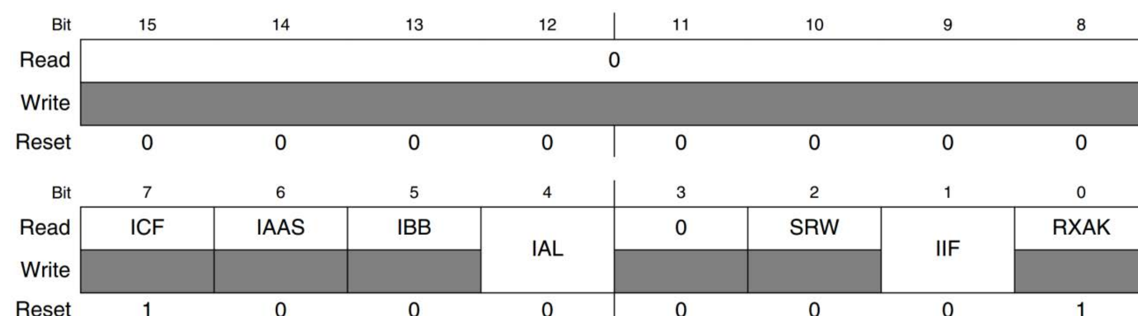


图 26.1.2.5 寄存器 I2Cx_I2SR 结构

寄存器 I2Cx_I2SR 的各位含义如下：

ICF(bit7)：数据传输状态位，为 0 的时候表示数据正在传输，为 1 的时候表示数据传输完成。

IAAS(bit6)：当为 1 的时候表示 I2C 地址，也就是 I2Cx_IADR 寄存器中的地址是从设备地址。

IBB(bit5)：I2C 总线忙标志位，当为 0 的时候表示 I2C 总线空闲，为 1 的时候表示 I2C 总线忙。

IAL(bit4)：仲裁丢失位，为 1 的时候表示发生仲裁丢失。

SRW(bit3)：从机读写状态位，当 I2C 作为从机的时候使用，此位用来表明主机发送给从机的是读还是写命令。为 0 的时候表示主机要向从机写数据，为 1 的时候表示主机要从从机读取数据。

IIF(bit1)：I2C 中断挂起标志位，当为 1 的时候表示有中断挂起，此位需要软件清零。

RXAK(bit0)：应答信号标志位，为 0 的时候表示接收到 ACK 应答信号，为 1 的话表示检测到 NO ACK 信号。

最后一个寄存器就是 I2Cx_I2DR，这是 I2C 的数据寄存器，此寄存器只有低 8 位有效，当要发送数据的时候将要发送的数据写入到此寄存器，如果要接收数据的话直接读取此寄存器即可得到接收到的数据。

关于 I2C 的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 1462 页的 31.7 小节。

26.1.3 AP3216C 简介

I.MX6U-ALPHA 开发板上通过 I2C1 连接了一个三合一环境传感器：AP3216C，AP3216C 是由敦南可以推出的一款传感器，其支持环境光强度(ALS)、接近距离(PS)和红外线强度(IR)这三个环境参数检测。该芯片可以通过 IIC 接口与主控制相连，并且支持中断，AP3216C 的特点如下：

①、I2C 接口，快速模式下波特率可以到 400Kbit/S

②、多种工作模式选择：ALS、PS+IR、ALS+PS+IR、PD 等等。

③、内建温度补偿电路。

④、宽工作温度范围(-30°C～+80°C)。

⑤、超小封装，4.1mm x 2.4mm x 1.35mm

⑥、环境光传感器具有 16 为分辨率。

⑦、接近传感器和红外传感器具有 10 为分辨率。

AP3216C 常被用于手机、平板、导航设备等，其内置的接近传感器可以用于检测是否有物体接近，比如手机上用来检测耳朵是否接触听筒，如果检测到的话就表示正在打电话，手机就会关闭手机屏幕以省电。也可以使用环境光传感器检测光照强度，可以实现自动背光亮度调节。AP3216C 结构如图 26.1.3.1 所示：

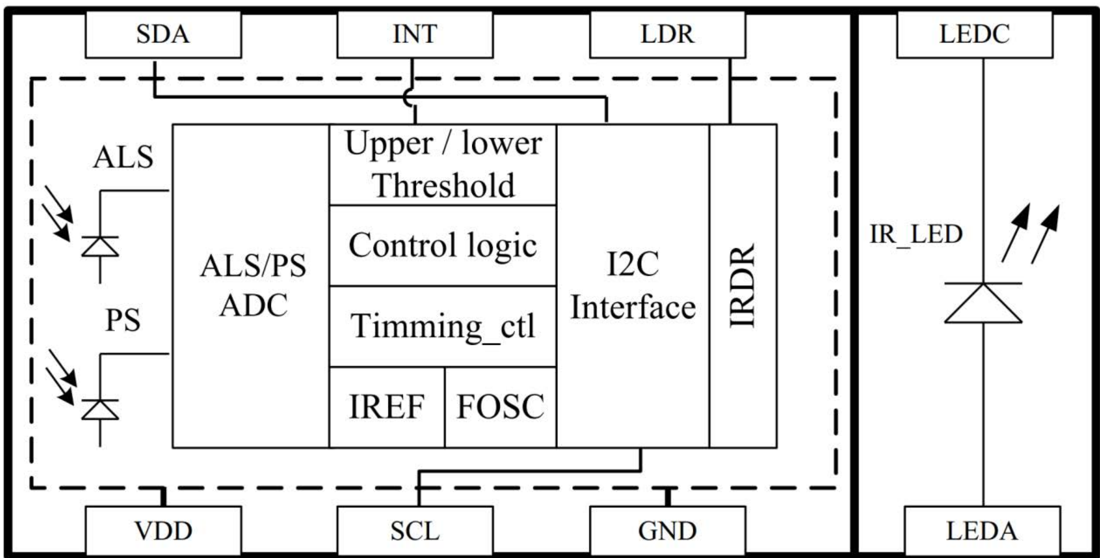


图 26.1.3.1 AP3216C 结构图

AP3216 的设备地址为 0X1E，同几乎所有的 I2C 从器件一样，AP3216C 内部也有一些寄存器，通过这些寄存器我们可以配置 AP3216C 的工作模式，并且读取相应的数据。AP3216C 我们用的寄存器如表 26.1.3.1 所示：

寄存器地址	位	寄存器功能	描述
0X00	2:0	系统模式	000：掉电模式(默认)。 001：使能 ALS。

			010：使能 PS+IR。 011：使能 ALS+PS+IR。 100：软复位。 101：ALS 单次模式。 110：PS+IR 单次模式。 111：ALS+PS+IR 单次模式。
0X0A	7	IR 低位数据	0：IR&PS 数据有效，1:无效
	1:0		IR 最低 2 位数据。
0X0B	7:0	IR 高位数据	IR 高 8 位数据。
0X0C	7:0	ALS 低位数据	ALS 低 8 位数据。
0X0D	7:0	ALS 高位数据	ALS 高 8 位数据。
0X0E	7	PS 低位数据	0，物体在远离；1，物体在接近。
	6		0，IR&PS 数据有效；1，IR&PS 数据无效
	3:0		PS 最低 4 位数据。
0X0F	7	PS 高位数据	0，物体在远离；1，物体在接近。
	6		0，IR&PS 数据有效；1，IR&PS 数据无效
	5:0		PS 最低 6 位数据。

表 26.1.3.1 本章使用的 AP3216C 寄存器表

在表 26.1.3.1 中，0X00 这个寄存器是模式控制寄存器，用来设置 AP3216C 的工作模式，一般开始先将其设置为 0X04，也就是先软件复位一次 AP3216C。接下来根据实际使用情况选择合适的工作模式，比如设置为 0X03，也就是开启 ALS+PS+IR。从 0X0A~0X0F 这 6 个寄存器就是数据寄存器，保存着 ALS、PS 和 IR 这三个传感器获取到的数据值。如果同时打开 ALS、PS 和 IR 的读取间隔最少要 112.5ms，因为 AP3216C 完成一次转换需要 112.5ms。关于 AP3216C 的介绍就到这里，如果要想详细的研究此芯片的话，请大家自行查阅其数据手册。

本章实验中我们通过 I.MX6U 的 I2C1 来读取 AP3216C 内部的 ALS、PS 和 IR 这三个传感器的值，并且在 LCD 上显示。开机会先检测 AP3216C 是否存在，一般的芯片是有个 ID 寄存器，通过读取 ID 寄存器判断 ID 是否正确就可以检测芯片是否存在。但是 AP3216C 没有 ID 寄存器，所以我们就通过向寄存器 0X00 写入一个值，然后再读取 0X00 寄存器，判断读出得到值和写入的是否相等，如果相等就表示 AP3216C 存在，否则的话 AP3216C 就不存在。本章的配置步骤如下：

1、初始化相应的 IO

初始化 I2C1 相应的 IO，设置其复用功能，如果要使用 AP3216C 中断功能的话，还需要设置 AP3216C 的中断 IO。

2、初始化 I2C1

初始化 I2C1 接口，设置波特率。

3、初始化 AP3216C

初始化 AP3216C，读取 AP3216C 的数据。

26.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、 RGB LCD 屏幕。

③、AP3216C

④、串口

AP3216C 是在 I.MX6U-ALPHA 开发板底板上，原理图如图 26.2.1 所示：

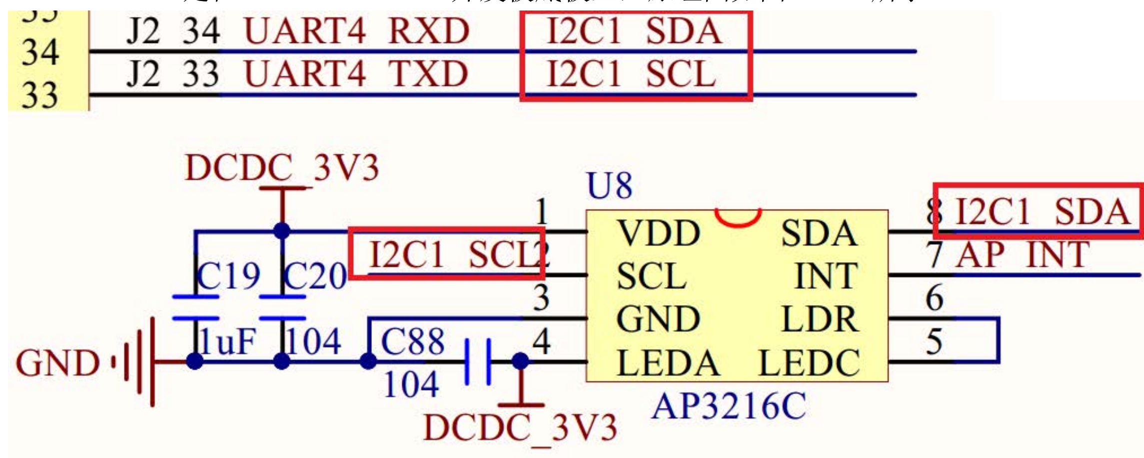


图 26.2.1 AP3216C 原理图

从图 26.2.1 可以看出 AP3216C 使用的是 I2C1，其中 I2C1_SCL 使用的 UART4_RXD 这个 IO、I2C1_SDA 使用的是 UART4_TXD 这个 IO。

26.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 17_i2c。

本章实验在上一章例程的基础上完成，更改工程名字为“ap3216c”，然后在 bsp 文件夹下创建名为“i2c”和“ap3216c”的文件。在 bsp/i2c 中新建 bsp_i2c.c 和 bsp_i2c.h 这两个文件，在 bsp/ap3216c 中新建 bsp_ap3216c.c 和 bsp_ap3216c.h 这两个文件。bsp_i2c.c 和 bsp_i2c.h 是 I.MX6U 的 I2C 文件，bsp_ap3216c.c 和 bsp_ap3216c.h 是 AP3216C 的驱动文件。在 bsp_i2c.h 中输入如下内容：

示例代码 26.3.1 bsp_i2c.h 文件代码

```
1  #ifndef _BSP_I2C_H
2  #define _BSP_I2C_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名      : bsp_i2c.h
6  作者        : 左忠凯
7  版本        : V1.0
8  描述        : IIC 驱动文件。
9  其他        : 无
10 论坛        : www.openedv.com
11 日志        : 初版 V1.0 2019/1/15 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 相关宏定义 */
16 #define I2C_STATUS_OK                 (0)
```

```
17 #define I2C_STATUS_BUSY             (1)
18 #define I2C_STATUS_IDLE             (2)
19 #define I2C_STATUS_NAK              (3)
20 #define I2C_STATUS_ARBITRATIONLOST  (4)
21 #define I2C_STATUS_TIMEOUT          (5)
22 #define I2C_STATUS_ADDRNAK          (6)
23 
24 /*
25  * I2C 方向枚举类型
26  */
27 enum i2c_direction
28 {
29     kI2C_Write = 0x0,                   /* 主机向从机写数据 */
30     kI2C_Read = 0x1,                    /* 主机从从机读数据 */
31 };
32 
33 /*
34  * 主机传输结构体
35  */
36 struct i2c_transfer
37 {
38     unsigned char slaveAddress;         /* 7 位从机地址     */
39     enum i2c_direction direction;       /* 传输方向         */
40     unsigned int subaddress;            /* 寄存器地址       */
41     unsigned char subaddressSize;       /* 寄存器地址长度   */
42     unsigned char *volatile data;       /* 数据缓冲区       */
43     volatile unsigned int dataSize;     /* 数据缓冲区长度   */
44 };
45 
46 /*
47  *函数声明
48  */
49 void i2c_init(I2C_Type *base);
50 unsigned char i2c_master_start(I2C_Type *base,
                                  unsigned char address,
                                  enum i2c_direction direction);
51 unsigned char i2c_master_repeated_start(I2C_Type *base,
                                  unsigned char address,
                                   enum i2c_direction direction);
52 unsigned char i2c_check_and_clear_error(I2C_Type *base,
                                  unsigned int status);
53 unsigned char i2c_master_stop(I2C_Type *base);
54 void i2c_master_write(I2C_Type *base, const unsigned char *buf,
```

```
                                    unsigned int size);
55 void i2c_master_read(I2C_Type *base, unsigned char *buf,
                                   unsigned int size);
56 unsigned char i2c_master_transfer(I2C_Type *base,
                                                    struct i2c_transfer *xfer);
57
58 #endif
```

第 16 到 22 行定义了一些 I2C 状态相关的宏。第 27 到 31 行定义了一个枚举类型 i2c_direction，此枚举类型用来表示 I2C 主机对从机的操作，也就是读数据还是写数据。第 36 到 44 行定义了一个结构体 i2c_transfer，此结构体用于 I2C 的数据传输。剩下的就是一些函数声明了，总体来说 bsp_i2c.h 文件里面的内容还是很简单的。接下来在文件 bsp_i2c.c 里面输入如下内容：

示例代码 26.3.2 bsp_i2c.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_i2c.c
作者      : 左忠凯
版本      : V1.0
描述      : IIC 驱动文件。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1   #include "bsp_i2c.h"
2   #include "bsp_delay.h"
3   #include "stdio.h"
4
5   /*
6    * @description   : 初始化 I2C，波特率 100KHZ
7    * @param - base  : 要初始化的 IIC 设置
8    * @return        : 无
9    */
10  void i2c_init(I2C_Type *base)
11  {
12      /* 1、配置 I2C */
13      base->I2CR &= ~(1 << 7); /* 要访问 I2C 的寄存器，首先需要先关闭 I2C */
14
15      /* 设置波特率为 100K
16       * I2C 的时钟源来源于 IPG_CLK_ROOT=66Mhz
17       * IFDR 设置为 0X15，也就是 640 分频，
18       * 66000000/640=103.125KHz≈100KHz。
19       */
20      base->IFDR = 0X15 << 0;
21
```

```
22     /* 设置寄存器 I2CR，开启 I2C */
23     base->I2CR |= (1<<7);
24 }
25
26 /*
27  * @description          : 发送重新开始信号
28  * @param - base         : 要使用的 IIC
29  * @param - addrss       : 设备地址
30  * @param - direction    : 方向
31  * @return               : 0 正常 其他值 出错
32  */
33 unsigned char i2c_master_repeated_start(I2C_Type *base,
                                       unsigned char address,
                                       enum i2c_direction direction)
34 {
35     /* I2C 忙并且工作在从模式,跳出 */
36     if(base->I2SR & (1 << 5) && (((base->I2CR) & (1 << 5)) == 0))
37         return 1;
38
39     /*
40      * 设置寄存器 I2CR
41      * bit[4]: 1 发送
42      * bit[2]: 1 产生重新开始信号
43      */
44     base->I2CR |=  (1 << 4) | (1 << 2);
45
46     /*
47      * 设置寄存器 I2DR，bit[7:0] : 要发送的数据，这里写入从设备地址
48      */
49     base->I2DR = ((unsigned int)address << 1) |
                  ((direction == kI2C_Read)? 1 : 0);
50     return 0;
51 }
52
53 /*
54  * @description          : 发送开始信号
55  * @param - base         : 要使用的 IIC
56  * @param - addrss       : 设备地址
57  * @param - direction    : 方向
58  * @return               : 0 正常 其他值 出错
59  */
60 unsigned char i2c_master_start(I2C_Type *base,
                              unsigned char address,
```

```
                                   enum i2c_direction direction)
61  {
62      if(base->I2SR & (1 << 5))          /* I2C 忙 */
63          return 1;
64  
65      /*
66       * 设置寄存器 I2CR
67       * bit[5]: 1 主模式
68       * bit[4]: 1 发送
69       */
70      base->I2CR |=  (1 << 5) | (1 << 4);
71  
72      /*
73       * 设置寄存器 I2DR，bit[7:0] : 要发送的数据，这里写入从设备地址
74       */
75      base->I2DR = ((unsigned int)address << 1) | 
                    ((direction == kI2C_Read)? 1 : 0);
76      return 0;
77  }
78  
79  /*
80   * @description      : 检查并清除错误
81   * @param - base     : 要使用的 IIC
82   * @param - status   : 状态
83   * @return           : 状态结果
84   */
85  unsigned char i2c_check_and_clear_error(I2C_Type *base, 
                                            unsigned int status)
86  {
87      if(status & (1<<4))             /* 检查是否发生仲裁丢失错误  */
88      {
89          base->I2SR &= ~(1<<4);      /* 清除仲裁丢失错误位        */
90          base->I2CR &= ~(1 << 7);    /* 先关闭 I2C                */
91          base->I2CR |= (1 << 7);     /* 重新打开 I2C              */
92          return I2C_STATUS_ARBITRATIONLOST;
93      }
94      else if(status & (1 << 0))      /* 没有接收到从机的应答信号  */
95      {
96          return I2C_STATUS_NAK;      /* 返回 NAK(No acknowledge) */
97      }
98      return I2C_STATUS_OK;
99  }
100
```

```
101 /*
102  * @description      : 停止信号
103  * @param - base      : 要使用的 IIC
104  * @param            : 无
105  * @return           : 状态结果
106  */
107 unsigned char i2c_master_stop(I2C_Type *base)
108 {
109     unsigned short timeout = 0XFFFF;
110 
111     /* 清除 I2CR 的 bit[5:3]这三位 */
112     base->I2CR &= ~((1 << 5) | (1 << 4) | (1 << 3));
113     while((base->I2SR & (1 << 5))) /* 等待忙结束   */
114     {
115         timeout--;
116         if(timeout == 0)           /* 超时跳出     */
117             return I2C_STATUS_TIMEOUT;
118     }
119     return I2C_STATUS_OK;
120 }
121 
122 /*
123  * @description      : 发送数据
124  * @param - base      : 要使用的 IIC
125  * @param - buf       : 要发送的数据
126  * @param - size      : 要发送的数据大小
127  * @param - flags     : 标志
128  * @return           : 无
129  */
130 void i2c_master_write(I2C_Type *base, const unsigned char *buf, 
                              unsigned int size)
131 {
132     while(!(base->I2SR & (1 << 7)));       /* 等待传输完成 */
133     base->I2SR &= ~(1 << 1);               /* 清除标志位   */
134     base->I2CR |= 1 << 4;                  /* 发送数据     */
135     while(size--)
136     {
137         base->I2DR = *buf++;    /* 将 buf 中的数据写入到 I2DR 寄存器 */
138         while(!(base->I2SR & (1 << 1)));   /* 等待传输完成 */
139         base->I2SR &= ~(1 << 1);           /* 清除标志位   */
140 
141         /* 检查 ACK */
142         if(i2c_check_and_clear_error(base, base->I2SR))
```

```
143             break;
144     }
145     base->I2SR &= ~(1 << 1);
146     i2c_master_stop(base);                      /* 发送停止信号 */
147 }
148
149 /*
150  * @description       : 读取数据
151  * @param - base      : 要使用的 IIC
152  * @param - buf       : 读取到数据
153  * @param - size      : 要读取的数据大小
154  * @return            : 无
155  */
156 void i2c_master_read(I2C_Type *base, unsigned char *buf,
                         unsigned int size)
157 {
158     volatile uint8_t dummy = 0;
159
160     dummy++;                                /* 防止编译报错     */
161     while(!(base->I2SR & (1 << 7)));        /* 等待传输完成     */
162     base->I2SR &= ~(1 << 1);                /* 清除中断挂起位   */
163     base->I2CR &= ~((1 << 4) | (1 << 3));   /* 接收数据       */
164     if(size == 1)     /* 如果只接收一个字节数据的话发送 NACK 信号 */
165        base->I2CR |= (1 << 3);
166
167     dummy = base->I2DR;                     /* 假读             */
168     while(size--)
169     {
170        while(!(base->I2SR & (1 << 1)));     /* 等待传输完成 */
171        base->I2SR &= ~(1 << 1);             /* 清除标志位   */
172
173        if(size == 0)
174            i2c_master_stop(base);           /* 发送停止信号 */
175        if(size == 1)
176            base->I2CR |= (1 << 3);
177        *buf++ = base->I2DR;
178     }
179 }
180
181 /*
182  * @description   : I2C 数据传输，包括读和写
183  * @param - base  : 要使用的 IIC
184  * @param - xfer  : 传输结构体
```

```
185  * @return        : 传输结果,0 成功，其他值 失败;
186  */
187 unsigned char i2c_master_transfer(I2C_Type *base,
                                      struct i2c_transfer *xfer)
188 {
189     unsigned char ret = 0;
190     enum i2c_direction direction = xfer->direction;
191 
192     base->I2SR &= ~((1 << 1) | (1 << 4));     /* 清除标志位   */
193     while(!((base->I2SR >> 7) & 0X1)){};      /* 等待传输完成 */
194     /* 如果是读的话，要先发送寄存器地址，所以要先将方向改为写 */
195     if ((xfer->subaddressSize > 0) && (xfer->direction ==
                                                    kI2C_Read))
196         direction = kI2C_Write;
197     ret = i2c_master_start(base, xfer->slaveAddress, direction);
198     if(ret)
199         return ret;
200     while(!(base->I2SR & (1 << 1))){};     /* 等待传输完成 */
201     ret = i2c_check_and_clear_error(base, base->I2SR);
202     if(ret)
203     {
204         i2c_master_stop(base);             /* 发送出错，发送停止信号 */
205         return ret;
206     }
207 
208     /* 发送寄存器地址 */
209     if(xfer->subaddressSize)
210     {
211         do
212         {
213             base->I2SR &= ~(1 << 1);          /* 清除标志位   */
214             xfer->subaddressSize--;           /* 地址长度减一 */
215             base->I2DR =  ((xfer->subaddress) >> (8 *
                                 xfer->subaddressSize));
216             while(!(base->I2SR & (1 << 1)));  /* 等待传输完成 */
217             /* 检查是否有错误发生 */
218             ret = i2c_check_and_clear_error(base, base->I2SR);
219             if(ret)
220             {
221                 i2c_master_stop(base);        /* 发送停止信号 */
222                 return ret;
223             }
224         } while ((xfer->subaddressSize > 0) && (ret ==
```

```
                                                    I2C_STATUS_OK));
225
226         if(xfer->direction == kI2C_Read)       /* 读取数据      */
227         {
228             base->I2SR &= ~(1 << 1);           /* 清除中断挂起位 */
229             i2c_master_repeated_start(base, xfer->slaveAddress,
                                        kI2C_Read);
230             while(!(base->I2SR & (1 << 1))){}; /* 等待传输完成  */
231
232             /* 检查是否有错误发生 */
233             ret = i2c_check_and_clear_error(base, base->I2SR);
234             if(ret)
235             {
236                 ret = I2C_STATUS_ADDRNAK;
237                 i2c_master_stop(base);         /* 发送停止信号 */
238                 return ret;
239             }
240         }
241     }
242
243     /* 发送数据 */
244     if ((xfer->direction == kI2C_Write) && (xfer->dataSize > 0))
245         i2c_master_write(base, xfer->data, xfer->dataSize);
246     /* 读取数据 */
247     if ((xfer->direction == kI2C_Read) && (xfer->dataSize > 0))
248         i2c_master_read(base, xfer->data, xfer->dataSize);
249     return 0;
250 }
```

文件 bsp_i2c.c 中一共有 8 个函数，我们依次来看一下这些函数的功能，首先是函数 i2c_init，此函数用来初始化 I2C，重点是设置 I2C 的波特率，初始化完成以后开启 I2C。第 2 个函数是 i2c_master_repeated_start，此函数用来发送一个重复开始信号，发送开始信号的时候也会顺带发送从设备地址。第 3 个函数是 i2c_master_start，此函数用于发送一个开始信号，发送开始信号的时候也顺带发送从设备地址。第 4 个函数是 i2c_check_and_clear_error，此函数用于检查并清除错误。第 5 个函数是 i2c_master_stop，用于产生一个停止信号。第 6 和第 7 个函数分别为 i2c_master_write 和 i2c_master_read，这两个函数分别用于完成向 I2C 从设备写数据和从 I2C 从设备读数据。最后一个函数是 i2c_master_transfer，此函数就是用户最终调用的，用于完成 I2C 通信的函数，此函数会使用前面的函数拼凑出 I2C 读/写时序。此函数就是按照 26.1.1 小节讲解的 I2C 读写时序来编写的。

I2C 的操作函数已经准备好了，接下来就是使用前面编写 I2C 操作函数来配置 AP3216C 了，配置完成以后就可以读取 AP3216C 里面的传感器数据，在 bsp_ap3216c.h 输入如下所示内容：

示例代码 26.3.3 bsp_ap3216c.h 文件代码

```
1  #ifndef _BSP_AP3216C_H
2  #define _BSP_AP3216C_H
```

```
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_ap3216c.h
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : AP3216C 驱动头文件。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/3/26 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 #define AP3216C_ADDR          0X1E    /* AP3216C 器件地址    */
16 
17 /* AP3316C 寄存器 */
18 #define AP3216C_SYSTEMCONG    0x00    /* 配置寄存器          */
19 #define AP3216C_INTSTATUS     0X01    /* 中断状态寄存器      */
20 #define AP3216C_INTCLEAR      0X02    /* 中断清除寄存器      */
21 #define AP3216C_IRDATALOW     0x0A    /* IR 数据低字节       */
22 #define AP3216C_IRDATAHIGH    0x0B    /* IR 数据高字节       */
23 #define AP3216C_ALSDATALOW    0x0C    /* ALS 数据低字节      */
24 #define AP3216C_ALSDATAHIGH   0X0D    /* ALS 数据高字节      */
25 #define AP3216C_PSDATALOW     0X0E    /* PS 数据低字节       */
26 #define AP3216C_PSDATAHIGH    0X0F    /* PS 数据高字节       */
27 
28 /* 函数声明 */
29 unsigned char ap3216c_init(void);
30 unsigned char ap3216c_readonebyte(unsigned char addr,
                                    unsigned char reg);
31 unsigned char ap3216c_writeonebyte(unsigned char addr,
                                     unsigned char reg,
                                     unsigned char data);
32 void ap3216c_readdata(unsigned short *ir, unsigned short *ps,
                       unsigned short *als);
33 
34 #endif
```

第 45 到 26 行定义了一些宏，分别为 AP3216C 的设备地址和寄存器地址，剩下的就是函数声明。接下来在 bsp_ap3216c.c 中输入如下所示内容：

示例代码 26.3.4 bsp_ap3216c.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_ap3216c.c
作者      : 左忠凯
```

```
版本      : V1.0
描述      : AP3216C 驱动文件。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/3/26 左忠凯创建
****************************************************************/
1   #include "bsp_ap3216c.h"
2   #include "bsp_i2c.h"
3   #include "bsp_delay.h"
4   #include "cc.h"
5   #include "stdio.h"
6
7   /*
8    * @description   : 初始化 AP3216C
9    * @param         : 无
10   * @return        : 0 成功，其他值 错误代码
11   */
12  unsigned char ap3216c_init(void)
13  {
14      unsigned char data = 0;
15
16      /* 1、IO 初始化，配置 I2C IO 属性
17       * I2C1_SCL -> UART4_TXD
18       * I2C1_SDA -> UART4_RXD
19       */
20      IOMUXC_SetPinMux(IOMUXC_UART4_TX_DATA_I2C1_SCL, 1);
21      IOMUXC_SetPinMux(IOMUXC_UART4_RX_DATA_I2C1_SDA, 1);
22      IOMUXC_SetPinConfig(IOMUXC_UART4_TX_DATA_I2C1_SCL, 0x70B0);
23      IOMUXC_SetPinConfig(IOMUXC_UART4_RX_DATA_I2C1_SDA, 0X70B0);
24
25      /* 2、初始化 I2C1 */
26      i2c_init(I2C1);
27
28      /* 3、初始化 AP3216C   */
29      /* 复位 AP3216C        */
30      ap3216c_writeonebyte(AP3216C_ADDR, AP3216C_SYSTEMCONG, 0X04);
31      delayms(50); /* AP33216C 复位至少 10ms */
32
33      /* 开启 ALS、PS+IR     */
34      ap3216c_writeonebyte(AP3216C_ADDR, AP3216C_SYSTEMCONG, 0X03);
35
36      /* 读取刚刚写进去的 0X03 */
37      data = ap3216c_readonebyte(AP3216C_ADDR, AP3216C_SYSTEMCONG);
```

```
38     if(data == 0X03)
39         return 0;   /* AP3216C 正常    */
40     else
41         return 1;   /* AP3216C 失败    */
42 }
43 
44 /*
45  * @description   : 向 AP3216C 写入数据
46  * @param - addr  : 设备地址
47  * @param - reg   : 要写入的寄存器
48  * @param - data  : 要写入的数据
49  * @return        : 操作结果
50  */
51 unsigned char ap3216c_writeonebyte(unsigned char addr,
                                     unsigned char reg,
                                     unsigned char data)
52 {
53     unsigned char status=0;
54     unsigned char writedata=data;
55     struct i2c_transfer masterXfer;
56 
57     /* 配置 I2C xfer 结构体 */
58     masterXfer.slaveAddress = addr;     /* 设备地址            */
59     masterXfer.direction = kI2C_Write; /* 写入数据            */
60     masterXfer.subaddress = reg;        /* 要写入的寄存器地址    */
61     masterXfer.subaddressSize = 1;      /* 地址长度一个字节      */
62     masterXfer.data = &writedata;       /* 要写入的数据         */
63     masterXfer.dataSize = 1;            /* 写入数据长度 1 个字节  */
64 
65     if(i2c_master_transfer(I2C1, &masterXfer))
66         status=1;
67 
68     return status;
69 }
70 
71 /*
72  * @description   : 从 AP3216C 读取一个字节的数据
73  * @param - addr  : 设备地址
74  * @param - reg   : 要读取的寄存器
75  * @return        : 读取到的数据。
76  */
77 unsigned char ap3216c_readonebyte(unsigned char addr,
                                    unsigned char reg)
```

```
78  {
79      unsigned char val=0;
80  
81      struct i2c_transfer masterXfer;
82      masterXfer.slaveAddress = addr;     /* 设备地址            */
83      masterXfer.direction = kI2C_Read;   /* 读取数据            */
84      masterXfer.subaddress = reg;        /* 要读取的寄存器地址  */
85      masterXfer.subaddressSize = 1;      /* 地址长度一个字节    */
86      masterXfer.data = &val;             /* 接收数据缓冲区      */
87      masterXfer.dataSize = 1;            /* 读取数据长度 1 个字节 */
88      i2c_master_transfer(I2C1, &masterXfer);
89  
90      return val;
91  }
92  
93  /*
94   * @description  : 读取 AP3216C 的原始数据，包括 ALS,PS 和 IR, 注意！如果
95   *               :同时打开 ALS,IR+PS 两次数据读取的时间间隔要大于 112.5ms
96   * @param - ir   : ir 数据
97   * @param - ps   : ps 数据
98   * @param - ps   : als 数据
99   * @return       : 无。
100  */
101 void ap3216c_readdata(unsigned short *ir, unsigned short *ps,
                      unsigned short *als)
102 {
103     unsigned char buf[6];
104     unsigned char i;
105 
106     /* 循环读取所有传感器数据 */
107     for(i = 0; i < 6; i++)
108     {
109         buf[i] = ap3216c_readonebyte(AP3216C_ADDR,
                                   AP3216C_IRDATALOW + i);
110     }
111 
112     if(buf[0] & 0X80)    /* IR_OF 位为 1,则数据无效   */
113         *ir = 0;
114     else                 /* 读取 IR 传感器的数据      */
115         *ir = ((unsigned short)buf[1] << 2) | (buf[0] & 0X03);
116 
117     *als = ((unsigned short)buf[3] << 8) | buf[2];/* 读取 ALS 数据 */
118 
```

```
119     if(buf[4] & 0x40)     /* IR_OF 位为 1,则数据无效  */
120         *ps = 0;
121     else                  /* 读取 PS 传感器的数据    */
122         *ps = ((unsigned short)(buf[5] & 0X3F) << 4) |
                   (buf[4] & 0X0F);
123 }
```

文件 bsp_ap3216c.c 里面共有 4 个函数，第 1 个函数是 ap3216c_init，顾名思义，此函数用于初始化 AP3216C，初始化成功的话返回 0，如果初始化失败就返回其他值。此函数先初始化所使用到的 IO，比如初始化 I2C1 的相关 IO，并设置其复用为 I2C1。然后此函数会调用 i2c_init 来初始化 I2C1，最后初始化 AP3216C。第 2 个和第 3 个函数分别为 ap3216c_writeonebyte 和 ap3216c_readonebyte，这两个函数分别是向 AP3216C 写入数据和从 AP3216C 读取数据。这两个函数都通过调用 bsp_i2c.c 中的函数 i2c_master_transfer 来完成对 AP3216C 的读写。最后一个函数就是 ap3216c_readdata，此函数用于读取 AP3216C 中的 ALS、PS 和 IR 传感器数据。

最后在 main.c 中输入如下代码：

示例代码 26.3.5 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    :   mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 18 IIC 实验
其他      : IIC 是最常用的接口，ALPHA 开发板上有多个 IIC 外设，本实验就
            来学习如何驱动 I.MX6U 的 IIC 接口，并且通过 IIC 接口读取板载
            AP3216C 的数据值。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/15 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "bsp_lcd.h"
9  #include "bsp_rtc.h"
10 #include "bsp_ap3216c.h"
11 #include "stdio.h"
12 
13 /*
14  * @description    : main 函数
15  * @param          : 无
16  * @return         : 无
```

```
17  */
18 int main(void)
19 {
20     unsigned short ir, als, ps;
21     unsigned char state = OFF;
22
23     int_init();                 /* 初始化中断(一定要最先调用！) */
24     imx6u_clkinit();            /* 初始化系统时钟              */
25     delay_init();               /* 初始化延时                  */
26     clk_enable();               /* 使能所有的时钟              */
27     led_init();                 /* 初始化 led                  */
28     beep_init();                /* 初始化 beep                 */
29     uart_init();                /* 初始化串口，波特率 115200    */
30     lcd_init();                 /* 初始化 LCD                  */
31
32     tftlcd_dev.forecolor = LCD_RED;
33     lcd_show_string(30, 50, 200, 16, 16,
                   (char*)"ALPHA-IMX6U IIC TEST");
34     lcd_show_string(30, 70, 200, 16, 16, (char*)"AP3216C TEST");
35     lcd_show_string(30, 90, 200, 16, 16, (char*)"ATOM@ALIENTEK");
36     lcd_show_string(30, 110, 200, 16, 16, (char*)"2019/3/26");
37
38     while(ap3216c_init())       /* 检测不到 AP3216C        */
39     {
40         lcd_show_string(30, 130, 200, 16, 16,
                       (char*)"AP3216C Check Failed!");
41         delayms(500);
42         lcd_show_string(30, 130, 200, 16, 16,
                       (char*)"Please Check!        ");
43         delayms(500);
44     }
45
46     lcd_show_string(30, 130, 200, 16, 16, (char*)"AP3216C Ready!");
47     lcd_show_string(30, 160, 200, 16, 16, (char*)" IR:");
48     lcd_show_string(30, 180, 200, 16, 16, (char*)" PS:");
49     lcd_show_string(30, 200, 200, 16, 16, (char*)"ALS:");
50     tftlcd_dev.forecolor = LCD_BLUE;
51     while(1)
52     {
53         ap3216c_readdata(&ir, &ps, &als);      /* 读取数据     */
54         lcd_shownum(30 + 32, 160, ir, 5, 16); /* 显示 IR 数据  */
55         lcd_shownum(30 + 32, 180, ps, 5, 16); /* 显示 PS 数据  */
56         lcd_shownum(30 + 32, 200, als, 5, 16);/* 显示 ALS 数据 */
```

```
57          delayms(120);
58          state = !state;
59          led_switch(LED0,state);
60      }
61      return 0;
62 }
```

第 38 行调用 ap3216c_init 来初始化 AP3216C，如果 AP3216C 初始化失败的话就会进入循环，会在 LCD 上不断的闪烁字符串"AP3216C Check Failed!"和"Please Check!"，直到 AP3216C 初始化成功。

第 53 行调用函数 ap3216c_readdata 来获取 AP3216C 的 ALS、PS 和 IR 传感器数据值，获取完成以后就会在 LCD 上显示出来。

文件 main.c 里面的内容总体上还是很简单的，实验程序的编写就到这里。

26.4 编译下载验证

26.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 ap3216c，然后在在 INCDIRS 和 SRCDIRS 中加入"bsp/i2c"和"bsp/ap3216c"，修改后的 Makefile 如下：

示例代码 26.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= ap3216c
3  
4  /* 省略掉其它代码...... */
5  
6  INCDIRS        :=  imx6ul \
7                     stdio/include \
8                     bsp/clk \
9                     bsp/led \
10                    bsp/delay  \
11                    bsp/beep \
12                    bsp/gpio \
13                    bsp/key \
14                    bsp/exit \
15                    bsp/int \
16                    bsp/epittimer \
17                    bsp/keyfilter \
18                    bsp/uart \
19                    bsp/lcd \
20                    bsp/rtc \
21                    bsp/i2c \
22                    bsp/ap3216c
23 
24 SRCDIRS        :=  project \
```

```
25                     stdio/lib \
26                     bsp/clk \
27                     bsp/led \
28                     bsp/delay \
29                     bsp/beep \
30                     bsp/gpio \
31                     bsp/key \
32                     bsp/exit \
33                     bsp/int \
34                     bsp/epittimer \
35                     bsp/keyfilter \
36                     bsp/uart \
37                     bsp/lcd \
38                     bsp/rtc \
39                     bsp/i2c \
40                     bsp/ap3216c
41 
42 /* 省略掉其它代码...... */
43 
44 clean:
45  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“ap3216c”，也就是目标名称为“ap3216c”。

第 21 和 22 行在变量 INCDIRS 中添加 I2C 和 AP3216C 的驱动头文件(.h)路径。

第 39 和 40 行在变量 SRCDIRS 中添加 I2C 和 AP3216C 驱动文件(.c)路径。

链接脚本保持不变。

26.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 ap3216c.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload ap3216c.bin /dev/sdd       //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。程序运行以后 LCD 界面如图 26.4.2.1 所示：

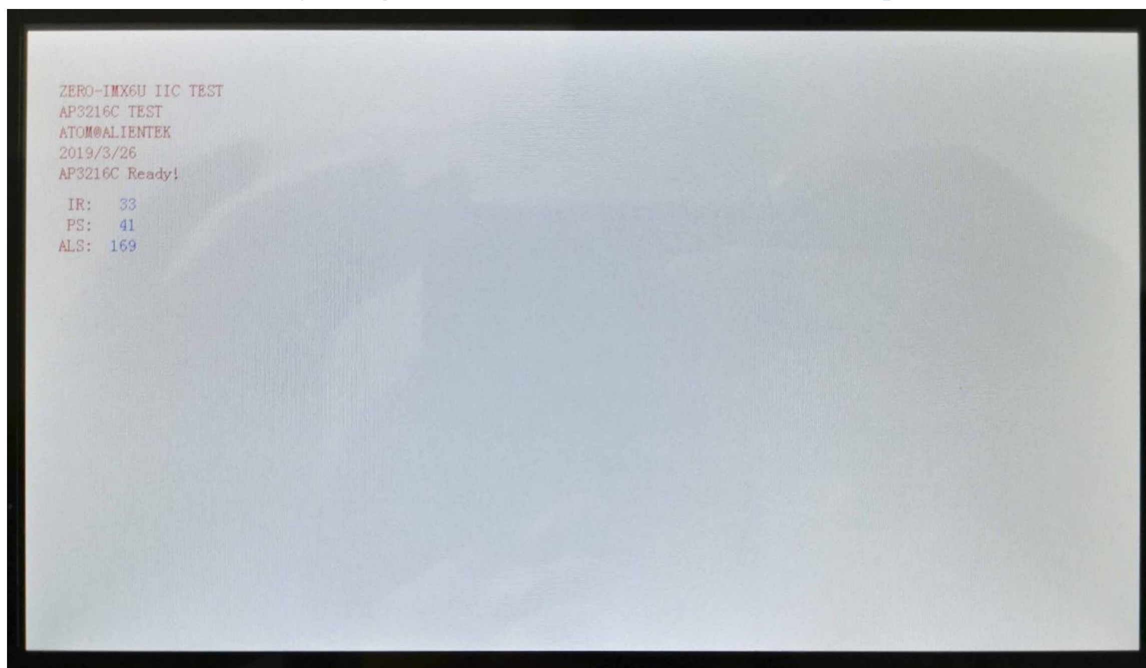


图 26.4.2.1 LCD 显示界面

图 26.4.2.1 中显示出了 AP3216C 的三个传感器的数据，大家可以用手遮住或者靠近 AP3216C，LCD 上的三个数据就会变化。

第二十七章 SPI 实验

同 I2C 一样，SPI 是很常用的通信接口，也可以通过 SPI 来连接众多的传感器。相比 I2C 接口，SPI 接口的通信速度很快，I2C 最多 400KHz，但是 SPI 可以到达几十 MHz。I.MX6U 也有 4 个 SPI 接口，可以通过这 4 个 SPI 接口来连接一些 I2C 外设。I.MX6U-ALPHA 使用 SPI3 接口连接了一个六轴传感器 ICM-20608，本章我们就来学习如何使用 I.MX6U 的 SPI 接口来驱动 ICM-20608，读取 ICM-20608 的六轴数据。

27.1 SPI & ICM-20608 简介

27.1.1 SPI 简介

上一章我们讲解了 I2C，I2C 是串行通信的一种，只需要两根线就可以完成主机和从机之间的通信，但是 I2C 的速度最高只能到 400KHz，如果对于访问速度要求比价高的话 I2C 就不适合了。本章我们就来学习一下另外一个和 I2C 一样广泛使用的串行通信：SPI，SPI 全称是 Serial Perripheral Interface，也就是串行外围设备接口。SPI 是 Motorola 公司推出的一种同步串行接口技术，是一种高速、全双工的同步通信总线，SPI 时钟频率相比 I2C 要高很多，最高可以工作在上百 MHz。SPI 以主从方式工作，通常是有一个主设备和一个或多个从设备，一般 SPI 需要 4 根线，但是也可以使用三根线(单向传输)，本章我们讲解标准的 4 线 SPI，这四根线如下：

①、CS/SS，Slave Select/Chip Select，这个是片选信号线，用于选择需要进行通信的从设备。I2C 主机是通过发送从机设备地址来选择需要进行通信的从机设备的，SPI 主机不需要发送从机设备，直接将相应的从机设备片选信号拉低即可。

②、SCK，Serial Clock，串行时钟，和 I2C 的 SCL 一样，为 SPI 通信提供时钟。

③、MOSI/SDO，Master Out Slave In/Serial Data Output，简称主出从入信号线，这根数据线只能用于主机向从机发送数据，也就是主机输出，从机输入。

④、MISO/SDI，Master In Slave Out/Serial Data Input，简称主入从出信号线，这根数据线只能用户从机向主机发送数据，也就是主机输入，从机输出。

SPI 通信都是由主机发起的，主机需要提供通信的时钟信号。主机通过 SPI 线连接多个从设备的结构如图 27.1.1.1 所示：

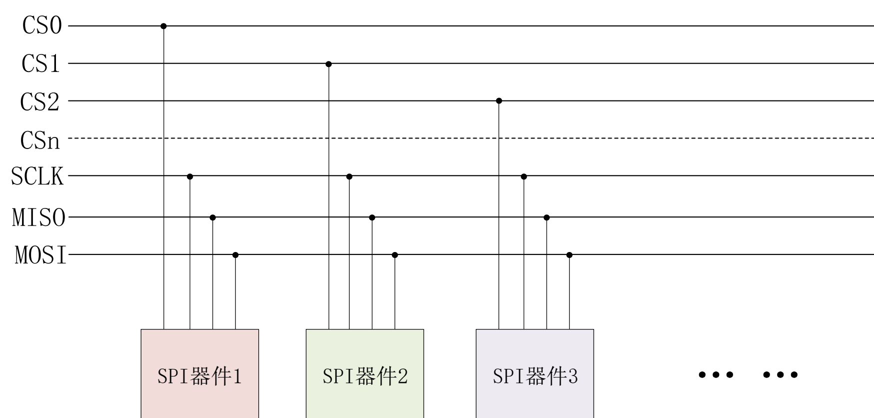


图 27.1.1.1 SPI 设备连接图

SPI 有四种工作模式，通过串行时钟极性(CPOL)和相位(CPHA)的搭配来得到四种工作模式：

①、CPOL=0，串行时钟空闲状态为低电平。

②、CPOL=1，串行时钟空闲状态为高电平，此时可以通过配置时钟相位(CPHA)来选择具体的传输协议。

③、CPHA=0，串行时钟的第一个跳变沿(上升沿或下降沿)采集数据。

④、CPHA=1，串行时钟的第二个跳变沿(上升沿或下降沿)采集数据。

这四种工作模式如图 27.1.1.2 所示：

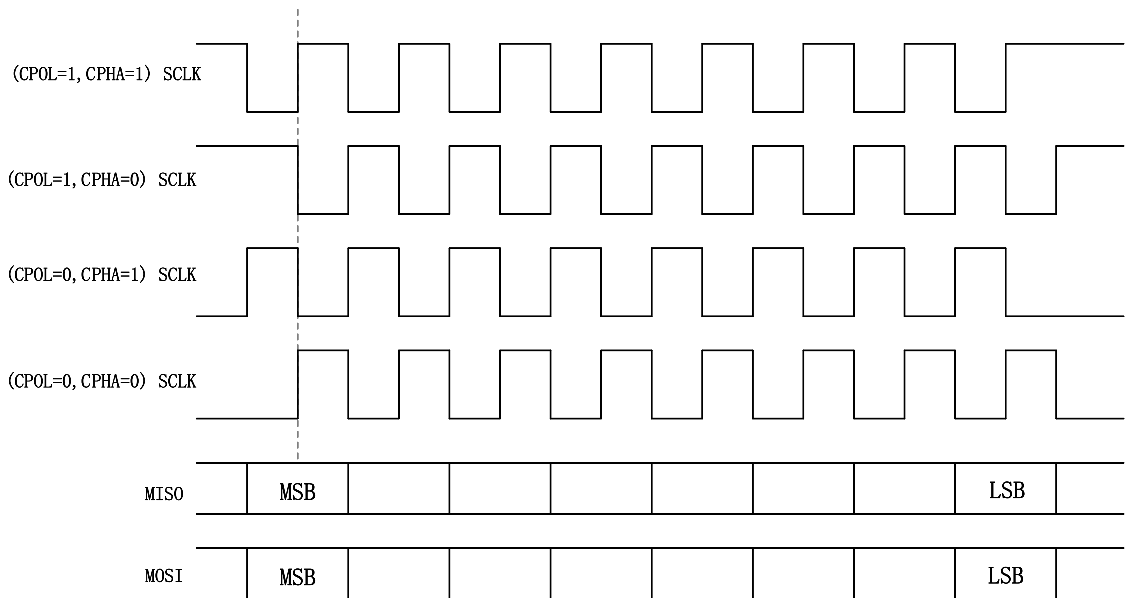


图 27.1.1.2 SPI 四种工作模式

根 I2C 一样，SPI 也是有时序图的，以 CPOL=0，CPHA=0 这个工作模式为例，SPI 进行全双工通信的时序如图 27.1.1.3 所示：

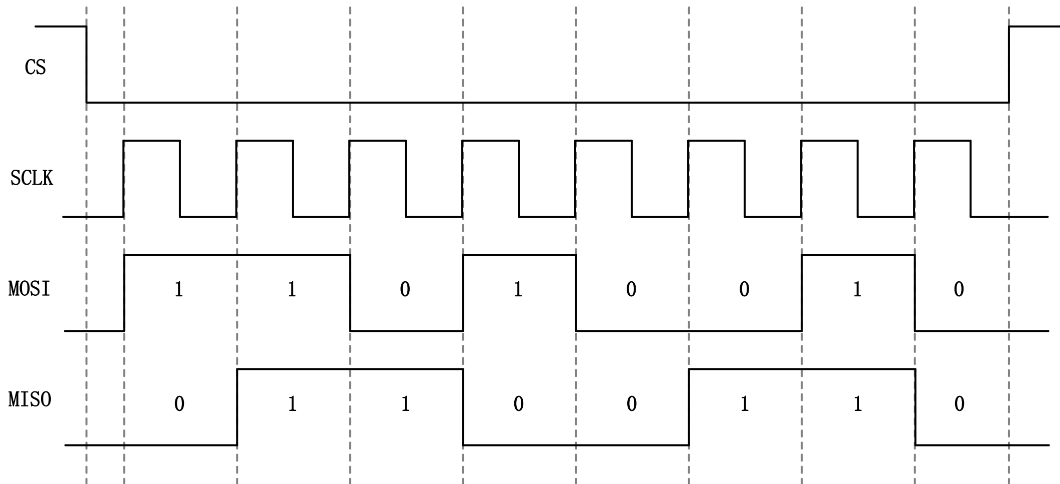


图 27.1.1.3 SPI 时序图

从图 27.1.1.3 可以看出，SPI 的时序图很简单，不像 I2C 那样还要分为读时序和写时序，因为 SPI 是全双工的，所以读写时序可以一起完成。图 27.1.1.3 中，CS 片选信号先拉低，选中要通信的从设备，然后通过 MOSI 和 MISO 这两根数据线进行收发数据，MOSI 数据线发出了 0XD2 这个数据给从设备，同时从设备也通过 MISO 线给主设备返回了 0X66 这个数据。这个就是 SPI 时序图。

关于 SPI 就讲解到这里，接下来我们看一下 I.MX6U 自带的 SPI 外设：ECSPI。

27.1.2 I.MX6U ECSPI 简介

I.MX6U 自带的 SPI 外设叫做 ECSPI，全称是 Enhanced Configurable Serial Peripheral Interface，别看前面加了个“EC”就以为和标准 SPI 有啥不同的，起始就是 SPI。ECSPI 有 64x32 个接收 FIFO(RXFIFO)和 64x32 个发送 FIFO(TXFIFO)，ECSPI 特性如下：

①、全双工同步串行接口。

②、可配置的主/从模式。

③、四个片选信号，支持多从机。

④、发送和接收都有一个 32x64 的 FIFO。

⑤、片选信号 SS/CS，时钟信号 SCLK 极性可配置。

⑥、支持 DMA。

I.MX6U 的 ECSPI 可以工作在主模式或从模式，本章我们使用主模式，I.MX6U 有 4 个 ECSPI，每个 ECSPI 支持四个片选信号，也就说，如果你要使用 ECSPI 的硬件片选信号的话，一个 ECSPI 可以支持 4 个外设。如果不使用硬件的片选信号就可以支持无数个外设，本章实验我们不使用硬件片选信号，因为硬件片选信号只能使用指定的片选 IO，软件片选的话可以使用任意的 IO。

我们接下来看一下 ECSPI 的几个重要的寄存器，首先看一下 ECSPIx_CONREG(x=1~4)寄存器，这是 ECSPI 的配置寄存器，此寄存器结构如图 27.1.2.1 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R/W	BURST_LENGTH												CHANNEL_SELECT		DRCTL	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R/W	PRE_DIVIDER				POST_DIVIDER				CHANNEL_MODE				SMC	XCH	HT	EN
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 27.1.2.1 寄存器 ECSPIx_CONREG 结构

寄存器 ECSPIx_CONREG 各位含义如下：

BURST_LENGTH(bit31:24)：突发长度，设置 SPI 的突发传输数据长度，在一次 SPI 发送中最大可以发送 2^12bit 数据。可以设置 0X000~0XFFF，分别对应 1~2^12bit。我们一般设置突发长度为一个字节，也就是 8bit，BURST_LENGTH=7。

CHANNEL_SELECT(bit19:18)：SPI 通道选择，一个 ECSPI 有四个硬件片选信号，每个片选信号是一个硬件通道，虽然我们本章实验使用的软件片选，但是 SPI 通道还是要选择的。可设置为 0~3，分别对应通道 0~3。I.MX6U-ALPHA 开发板上的 ICM-20608 的片选信号接的是 ECSPI3_SS0，也就是 ECSPI3 的通道 0，所以本章实验设置为 0。

DRCTL(bit17:16)：SPI 的 SPI_RDY 信号控制位，用于设置 SPI_RDY 信号，为 0 的话不关心 SPI_RDY 信号；为 1 的话 SPI_RDY 信号为边沿触发；为 2 的话 SPI_DRY 是电平触发。

PRE_DIVIDER(bit15:12)：SPI 预分频，ECSPI 时钟频率使用两步来完成分频，此位设置的是第一步，可设置 0~15，分别对应 1~16 分频。

POST_DIVIDER(bit11:8)：SPI 分频值，ECSPI 时钟频率的第二步分频设置，分频值为 2^POST_DIVIDER。

CHANNEL_MODE(bit7:4)：SPI 通道主/从模式设置，CHANNEL_MODE[3:0]分别对应 SPI 通道 3~0，为 0 的话就是设置为从模式，如果为 1 的话就是主模式。比如设置为 0X01 的话就是设置通道 0 为主模式。

SMC(bit3)：开始模式控制，此位只能在主模式下起作用，为 0 的话通过 XCH 位来开启 SPI 突发访问，为 1 的话只要向 TXFIFO 写入数据就开启 SPI 突发访问。

XCH(bit2)：此位只在主模式下起作用，当 SMC 为 0 的话此位用来控制 SPI 突发访问的开启。

HT(bit1)：HT 模式是能位，I.MX6ULL 不支持。

EN(bit0)：SPI 使能位，为 0 的话关闭 SPI，为 1 的话使能 SPI。

接下来看一下寄存器 ECSPIx_CONFIGREG，这个也是 ECSPI 的控制寄存器，此寄存器结

构如图 27.1.2.2 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	Reserved			HT_LENGTH					SCLK_CTL				DATA_CTL				SS_POL				SS_CTL				SCLK_POL				SCLK_PHA			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 27.1.2.2 寄存器 ECSPIx_CONFIGREG 结构

寄存器 ECSPIx_CONREG 用到的重要位如下：

HT_LENGTH(bit28:24)：HT 模式下的消息长度设置，I.MX6ULL 不支持。

SCLK_CTL(bit23:20)：设置 SCLK 信号线空闲状态电平，SCLK_CTL[3:0]分别对应通道 3~0，为 0 的话 SCLK 空闲状态为低电平，为 1 的话 SCLK 空闲状态为高电平。

DATA_CTL(bit19:16)：设置 DATA 信号线空闲状态电平，DATA_CTL[3:0]分别对应通道 3~0，为 0 的话 DATA 空闲状态为高电平，为 1 的话 DATA 空闲状态为低电平。

SS_POL(bit15:12)：设置 SPI 片选信号极性设置，SS_POL[3:0]分别对应通道 3~0，为 0 的话片选信号低电平有效，为 1 的话片选信号高电平有效。

SCLK_POL(bit7:4)：SPI 时钟信号极性设置，也就是 CPOL，SCLK_POL[3:0]分别对应通道 3~0，为 0 的话 SCLK 高电平有效(空闲的时候为低电平)，为 1 的话 SCLK 低电平有效(空闲的时候为高电平)。

SCLK_PHA(bit3:0)：SPI 时钟相位设置，也就是 CPHA，SCLK_PHA[3:0]分别对应通道 3~0，为 0 的话串行时钟的第一个跳变沿(上升沿或下降沿)采集数据，为 1 的话串行时钟的第二个跳变沿(上升沿或下降沿)采集数据。

通过 SCLK_POL 和 SCLK_PHA 可以设置 SPI 的工作模式。

接下来看一下寄存器 ECSPIx_PERIODREG，这个是 ECSPI 的采样周期寄存器，此寄存器结构如图 27.1.2.3 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R W	Reserved										CSD_CTL					
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CSRC	SAMPLE_PERIOD														
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 27.1.2.3 寄存器 ECSPIx_PERIODREG 结构

寄存器 ECSPIx_PERIODREG 用到的重要位如下：

CSD_CTL(bit21:16)：片选信号延时控制位，用于设置片选信号和第一个 SPI 时钟信号之间的时间间隔，范围为 0~63。

CSRC(bit15)：SPI 时钟源选择，为 0 的话选择 SPI CLK 为 SPI 的时钟源，为 1 的话选择 32.768KHz 的晶振为 SPI 时钟源。我们一般选择 SPI CLK 作为 SPI 时钟源，SPI CLK 时钟来源如图 27.1.2.4 所示：

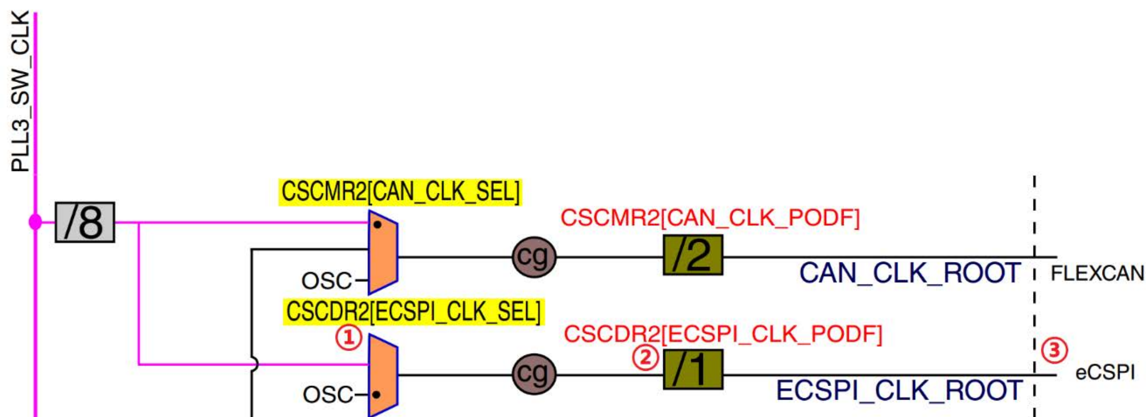


图 27.1.2.4 SPI CLK 时钟源

图 27.1.2.4 中各部分含义如下：

①、这是一个选择器，用于选择根时钟源，由寄存器 CSCDR2 的位 ECSPI_CLK_SEL 来控制，为 0 的话选择 pll3_60m 作为 ECSPI 根时钟源。为 1 的话选择 osc_clk 作为 ECSPI 时钟源。本章我们选择 pll3_60m 作为 ECSPI 根时钟源。

②、ECSPI 时钟分频值，由寄存器 CSCDR2 的位 ECSPI_CLK_PODF 开控制，分频值为 2^ECSPI_CLK_PODF。本章我们设置为 0，也就是 1 分频。

③、最终进入 ECSPI 的时钟，也就是 SPI CLK=60MHz。

SAMPLE_PERIO：采样周期寄存器，可设置为 0~0X7FFF 分别对应 0~32767 个周期。

接下来看一下寄存器 ECSPIx_STATREG，这个是 ECSPI 的状态寄存器，此寄存器结构如图 27.1.2.5 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	Reserved															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	Reserved								TC	RO	RF	RDR	RR	TF	TDR	TE
W									w1c	w1c						
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

图 27.1.2.5 寄存器 ECSPIx_STATREG 寄存器

寄存器 ECSPIx_STATREG 用到的重要位如下：

TC(bit7)：传输完成标志位，为 0 表示正在传输，为 1 表示传输完成。

RO(bit6)：RXFIFO 溢出标志位，为 0 表示 RXFIFO 无溢出，为 1 表示 RXFIFO 溢出。

RF(bit5)：RXFIFO 空标志位，为 0 表示 RXFIFO 不为空，为 1 表示 RXFIFO 为空。

RDR(bit4)：RXFIFO 数据请求标志位，此位为 0 表示 RXFIFO 里面的数据不大于 RX_THRESHOLD，此位为 1 的话表示 RXFIFO 里面的数据大于 RX_THRESHOLD。

RR(bit3)：RXFIFO 就绪标志位，为 0 的话 RXFIFO 没有数据，为 1 的话表示 RXFIFO 中至少有一个字的数据。

TF(bit2)：TXFIFO 满标志位，为 0 的话表示 TXFIFO 不为满，为 1 的话表示 TXFIFO 为满。

TDR(bit1)：TXFIFO 数据请求标志位，为 0 表示 TXFIFO 中的数据大于 TX_THRESHOLD，为 1 表示 TXFIFO 中的数据不大于 TX_THRESHOLD。

TE(bit0)：TXFIFO 空标志位，为 0 表示 TXFIFO 中至少有一个字的数据，为 1 表示 TXFIFO 为空。

最后就是两个数据寄存器，ECSPIx_TXDATA 和 ECSPIx_RXDATA，这两个寄存器都是 32 位的，如果要发送数据就向寄存器 ECSPIx_TXDATA 写入数据，读取及存取 ECSPIx_RXDATA 里面的数据就可以得到刚刚接收到的数据。

关于 ECSPI 的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 805 页的 20.7 小节。

27.1.3 ICM-20608 简介

ICM-20608 是 InvenSense 出品的一款 6 轴 MEMS 传感器，包括 3 轴加速度和 3 轴陀螺仪。ICM-20608 尺寸非常小，只有 3x3x0.75mm，采用 16P 的 LGA 封装。ICM-20608 内部有一个 512 字节的 FIFO。陀螺仪的量程范围可以编程设置，可选择±250，±500，±1000 和±2000°/s，加速度的量程范围也可以编程设置，可选择±2g，±4g，±4g，±8g 和±16g。陀螺仪和加速度计都是 16 位的 ADC，并且支持 I2C 和 SPI 两种协议，使用 I2C 接口的话通信速度最高可以达到 400KHz，使用 SPI 接口的话通信速度最高可达到 8MHz。I.MX6U-ALPHA 开发板上的 ICM-20608 通过 SPI 接口和 I.MX6U 连接在一起。ICM-20608 特性如下：

①、陀螺仪支持 X,Y 和 Z 三轴输出，内部集成 16 位 ADC，测量范围可设置：±250，±500，±1000 和±2000°/s。

②、加速度计支持 X,Y 和 Z 轴输出，内部集成 16 位 ADC，测量范围可设置：±2g，±4g，±4g，±8g 和±16g。

③、用户可编程中断。

④、内部包含 512 字节的 FIFO。

⑤、内部包含一个数字温度传感器。

⑥、耐 10000g 的冲击。

⑦、支持快速 I2C，速度可达 400KHz。

⑧、支持 SPI，速度可达 8MHz。

ICM-20608 的 3 轴方向如图 27.1.3.1 所示：

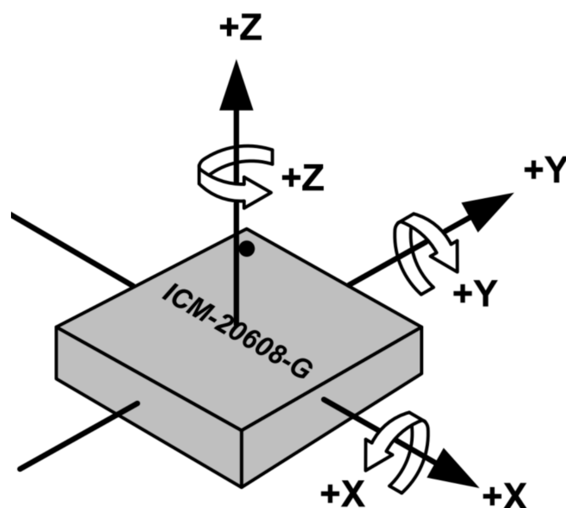


图 27.1.3.1 ICM-20608 检测轴方向和极性

ICM-20608 的结构框图如图 27.1.3.2 所示：

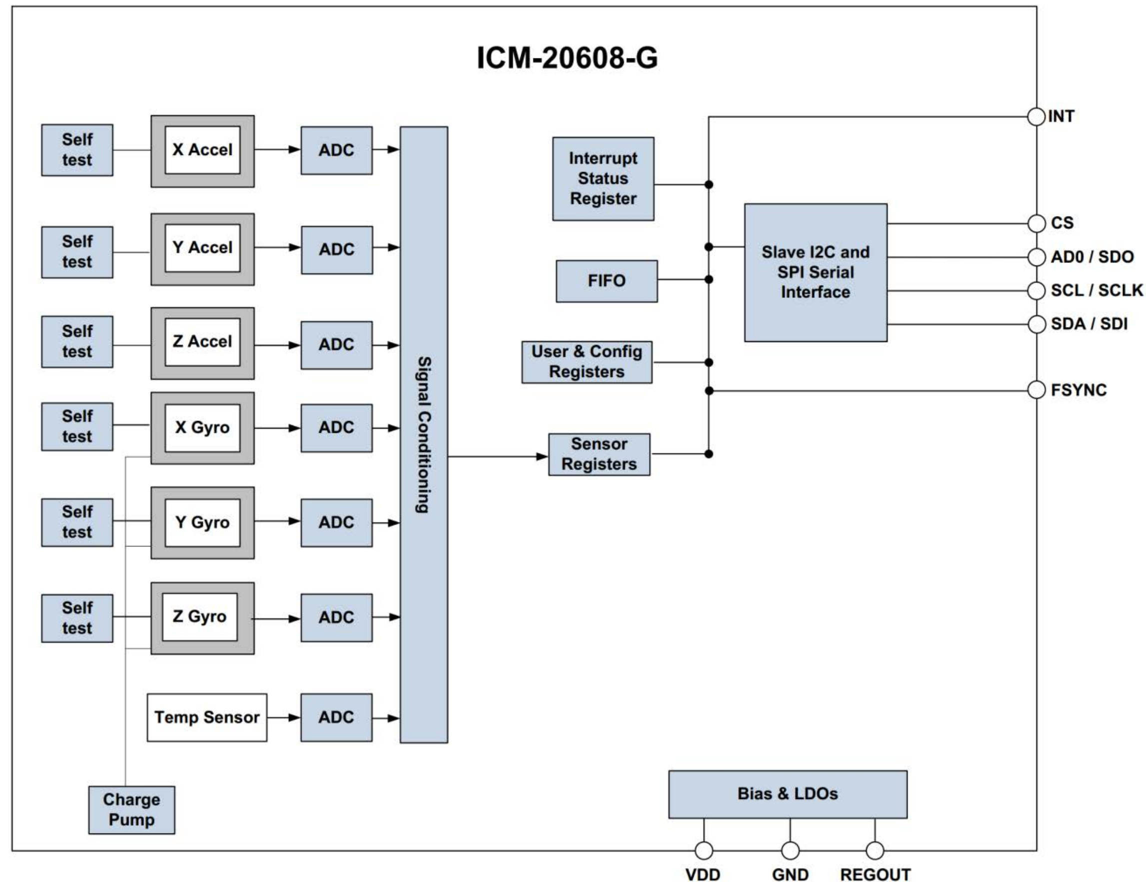


图 27.1.3.2 ICM-20608 框图

如果使用 IIC 接口的话 ICM-20608 的 AD0 引脚决定 I2C 设备从地址的最后一位，如果 AD0 为 0 的话 ICM-20608 从设备地址是 0X68，如果 AD0 为 1 的话 ICM-20608 从设备地址为 0X69。本章我们使用 SPI 接口，跟上一章使用 AP3216C 一样，ICM-20608 也是通过读写寄存器来配置和读取传感器数据，使用 SPI 接口读写寄存器需要 16 个时钟或者更多(如果读写操作包括多个字节的话)，第一个字节包含要读写的寄存器地址，寄存器地址最高位是读写标志位，如果是读的话寄存器地址最高位要为 1，如果是写的话寄存器地址最高位要为 0，剩下的 7 位才是实际的寄存器地址，寄存器地址后面跟着的就是读写的数据。表 27.1.3.1 列出了本章实验用到的一些寄存器和位，关于 ICM-20608 的详细寄存器和位的介绍请参考 ICM-20608 的寄存器手册：

寄存器地址	位	寄存器功能	描述
0X19	SMLPRT_DIV[7:0]	输出速率设置	设置输出速率，输出速率计算公式如下：SAMPLE_RATE=INTERNAL_SAMPLE_RATE/(1 + SMPLRT_DIV)
0X1A	DLPF_CFG[2:0]	芯片配置	设置陀螺仪低通滤波。可设置 0~7。
0X1B	FS_SEL[1:0]	陀螺仪量程设置	0：±250dps；1：±500dps；2：±1000dps 3：±2000dps
0X1C	ACC_FS_SEL[1:0]	加速度计量程设置	0：±2g；1：±4g；2：±8g；3：±16g
0X1D	A_DLPF_CFG[2:0]	加速度计低通滤波设置	设置加速度计的低通滤波，可设置 0~7。

0X1E	GYRO_CYCLE[7]	陀螺仪低功耗使能	0：关闭陀螺仪的低功耗功能。 1：使能陀螺仪的低功耗功能。
0X23	TEMP_FIFO_EN[7]	FIFO 使能控制	1：使能温度传感器 FIFO。 0：关闭温度传感器 FIFO。
	XG_FIFO_EN[6]		1：使能陀螺仪 X 轴 FIFO。 0：关闭陀螺仪 X 轴 FIFO。
	YG_FIFO_EN[5]		1：使能陀螺仪 Y 轴 FIFO。 0：关闭陀螺仪 Y 轴 FIFO。
	ZG_FIFO_EN[4]		1：使能陀螺仪 Z 轴 FIFO。 0：关闭陀螺仪 Z 轴 FIFO。
	ACCEL_FIFO_EN[3]		1：使能加速度计 FIFO。 0：关闭加速度计 FIFO。
0X3B	ACCEL_XOUT_H[7:0]	数据寄存器	加速度 X 轴数据高 8 位
0X3C	ACCEL_XOUT_L[7:0]		加速度 X 轴数据低 8 位
0X3D	ACCEL_YOUT_H[7:0]		加速度 Y 轴数据高 8 位
0X3E	ACCEL_YOUT_L[7:0]		加速度 Y 轴数据低 8 位
0X3F	ACCEL_ZOUT_H[7:0]		加速度 Z 轴数据高 8 位
0X40	ACCEL_ZOUT_L[7:0]		加速度 Z 轴数据低 8 位
0X41	TEMP_OUT_H[7:0]		温度数据高 8 位
0X42	TEMP_OUT_L[7:0]		温度数据低 8 位
0X43	GYRO_XOUT_H[7:0]		加速度计 X 轴数据高 8 位
0X44	GYRO_XOUT_L[7:0]		加速度计 X 轴数据低 8 位
0X45	GYRO_YOUT_H[7:0]		加速度计 Y 轴数据高 8 位
0X46	GYRO_YOUT_L[7:0]		加速度计 Y 轴数据低 8 位
0X47	GYRO_ZOUT_H[7:0]		加速度计 Z 轴数据高 8 位
0X48	GYRO_ZOUT_L[7:0]		加速度计 Z 轴数据低 8 位
0X6B	DEVICE_RESET[7]	电源管理寄存器 1	1：复位 ICM-20608。
	SLEEP[6]		0：退出休眠模式；1，进入休眠模式
0X6C	STBY_XA[5]	电源管理寄存器 2	0：使能加速度计 X 轴。 1：关闭加速度计 X 轴。
	STBY_YA[4]		0：使能加速度计 Y 轴。 1：关闭加速度计 Y 轴。
	STBY_ZA[3]		0：使能加速度计 Z 轴。 1：关闭加速度计 Z 轴。
	STBY_XG[2]		0：使能陀螺仪 X 轴。 1：关闭陀螺仪 X 轴。
	STBY_YG[1]		0：使能陀螺仪 Y 轴。 1：关闭陀螺仪 Y 轴。
	STBY_ZG[0]		0：使能陀螺仪 Z 轴。 1：关闭陀螺仪 Z 轴。
0X75	WHOAMI[7:0]		ID 寄存器，ICM-20608G 的 ID 为 0XAF，ICM-20608D 的 ID 为 0XAE。

表 27.1.3.1 ICM-20608 寄存器表

ICM-20608 的介绍就到这里，关于 ICM-20608 的详细介绍请参考 ICM-20608 的数据手册和寄存器手册。

27.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、 RGB LCD 屏幕。

③、ICM20608

④、串口

ICM-20608 是在 I.MX6U-ALPHA 开发板底板上，原理图如图 27.2.1 所示：

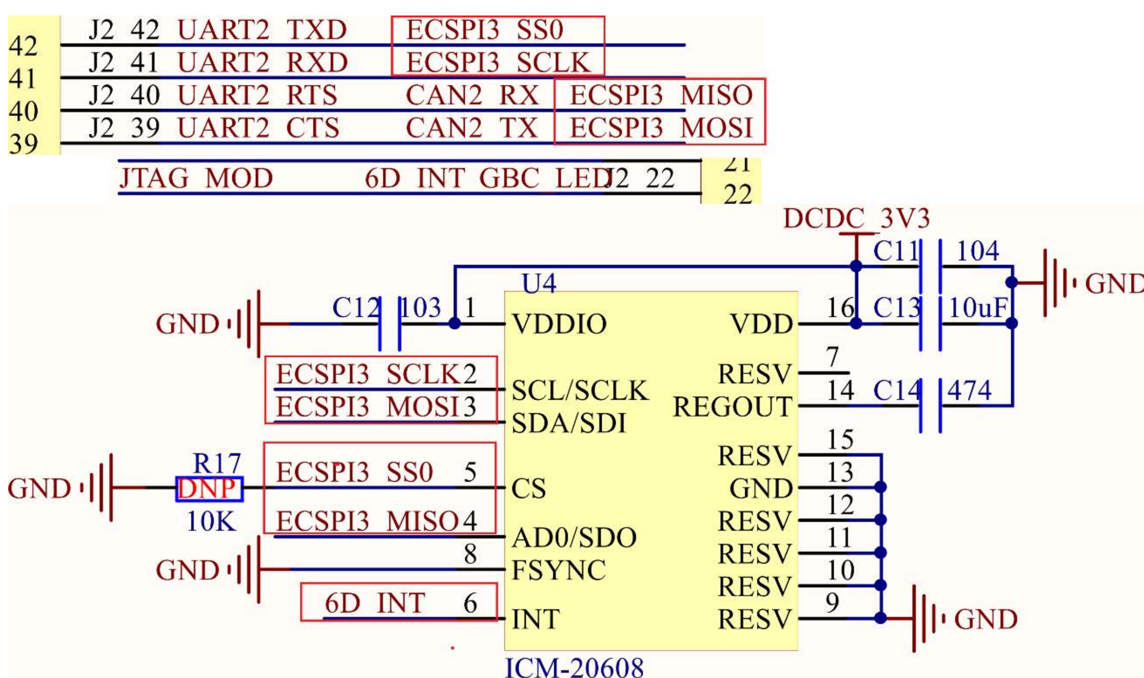


图 27.2.1 ICM-20608 原理图

27.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 18_spi。

本章实验在上一章例程的基础上完成，更改工程名字为“icm20608”，然后在 bsp 文件夹下创建名为“spi”和“icm20608”的文件。在 bsp/spi 中新建 bsp_spi.c 和 bsp_spi.h 这两个文件，在 bsp/icm20608 中新建 bsp_icm20608.c 和 bsp_icm20608.h 这两个文件。bsp_spi.c 和 bsp_spi.h 是 I.MX6U 的 SPI 文件，bsp_icm20608.c 和 bsp_icm20608.h 是 ICM20608 的驱动文件。在 bsp_spi.h 中输入如下内容：

示例代码 27.3.1 bsp_spi.h 文件代码

```
1  #ifndef _BSP_SPI_H
2  #define _BSP_SPI_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_spi.h
```

```
6  作者        : 左忠凯
7  版本        : V1.0
8  描述        : SPI 驱动头文件。
9  其他        : 无
10 论坛        : www.openedv.com
11 日志        : 初版 V1.0 2019/1/17 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 函数声明 */
16 void spi_init(ECSPI_Type *base);
17 unsigned char spich0_readwrite_byte(ECSPI_Type *base,
                                       unsigned char txdata);
18 #endif
```

文件 bsp_spi.h 内容很简单，就是函数声明。在文件 bsp_spi.c 中输入如下内容：

示例代码 27.3.2 bsp_spi.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : bsp_spi.c
作者      : 左忠凯
版本      : V1.0
描述      : SPI 驱动文件。
其他      : 无
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/17 左忠凯创建
***************************************************************/
1  #include "bsp_spi.h"
2  #include "bsp_gpio.h"
3  #include "stdio.h"
4  
5  /*
6   * @description    : 初始化 SPI
7   * @param - base   : 要初始化的 SPI
8   * @return         : 无
9   */
10 void spi_init(ECSPI_Type *base)
11 {
12     /* 配置 CONREG 寄存器
13      * bit0 :      1   使能 ECSPI
14      * bit3 :      1   当向 TXFIFO 写入数据以后立即开启 SPI 突发。
15      * bit[7:4]:   0001 SPI 通道 0 主模式，根据实际情况选择，开发板上的
16      *                  ICM-20608 接在 SS0 上，所以设置通道 0 为主模式
17      * bit[19:18]:00   选中通道 0(其实不需要，因为片选信号我们我们自己控制)
```

```
18       * bit[31:20]:0x7 突发长度为 8 个 bit。
19       */
20      ase->CONREG = 0;                       /* 先清除控制寄存器 */
21      ase->CONREG |= (1 << 0) | (1 << 3) | (1 << 4) | (7 << 20);
22
23      /*
24      * ECSPI 通道 0 设置,即设置 CONFIGREG 寄存器
25      * bit0:  0 通道 0 PHA 为 0
26      * bit4:  0 通道 0 SCLK 高电平有效
27      * bit8:  0 通道 0 片选信号 当 SMC 为 1 的时候此位无效
28      * bit12: 0 通道 0 POL 为 0
29      * bit16: 0 通道 0 数据线空闲时高电平
30      * bit20: 0 通道 0 时钟线空闲时低电平
31      */
32      base->CONFIGREG = 0;                /* 设置通道寄存器 */
33
34      /*
35      * ECSPI 通道 0 设置，设置采样周期
36      * bit[14:0] : 0X2000  采样等待周期，比如当 SPI 时钟为 10MHz 的时候
37      *                 0X2000 就等于 1/10000 * 0X2000 = 0.8192ms，也就是
38      *                 连续读取数据的时候每次之间间隔 0.8ms
39      * bit15  :     0  采样时钟源为 SPI CLK
40      * bit[21:16]: 0  片选延时，可设置为 0~63
41      */
42      base->PERIODREG = 0X2000;       /* 设置采样周期寄存器 */
43
44      /*
45      * ECSPI 的 SPI 时钟配置，SPI 的时钟源来源于 pll3_sw_clk/8=480/8=60MHz
46      * SPI CLK = (SourceCLK / PER_DIVIDER) / (2^POST_DIVEDER)
47      * 比如我们现在要设置 SPI 时钟为 6MHz，那么设置如下：
48      * PER_DIVIDER = 0X9。
49      * POST_DIVIDER = 0X0。
50      * SPI CLK = 60000000/(0X9 + 1) = 60000000=6MHz
51      */
52      base->CONREG &= ~((0XF << 12) | (0XF << 8)); /* 清除以前的设置 */
53      base->CONREG |= (0X9 << 12);          /* 设置 SPI CLK = 6MHz */
54  }
55
56  /*
57   * @description    : SPI 通道 0 发送/接收一个字节的数据
58   * @param - base   : 要使用的 SPI
59   * @param - txdata : 要发送的数据
60   * @return         : 无
```

```
61  */
62 unsigned char spich0_readwrite_byte(ECSPI_Type *base, 
                                       unsigned char txdata)
63 {
64     uint32_t  spirxdata = 0;
65     uint32_t  spitxdata = txdata;
66 
67     /* 选择通道 0 */
68     base->CONREG &= ~(3 << 18);
69     base->CONREG |= (0 << 18);
70 
71     while((base->STATREG & (1 << 0)) == 0){} /* 等待发送 FIFO 为空 */
72     base->TXDATA = spitxdata;
73 
74     while((base->STATREG & (1 << 3)) == 0){} /* 等待接收 FIFO 有数据 */
75     spirxdata = base->RXDATA;
76     return spirxdata;
77 }
```

文件 bsp_spi.c 中有两个函数：spi_init 和 spich0_readwrite_byte，函数 spi_init 是 SPI 初始化函数，此函数会初始化 SPI 的时钟，通道等。函数 spich0_readwrite_byte 是 SPI 收发函数，通过此函数即可完成 SPI 的全双工数据收发。

接下来在文件 bsp_icm20608.h 中输入如下内容：

示例代码 27.3.3 bsp_icm20608.h 文件代码

```
1  #ifndef _BSP_ICM20608_H
2  #define _BSP_ICM20608_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_icm20608.h
6  作者       : 左忠凯
7  版本       : V1.0
8  描述       : ICM20608 驱动文件。
9  其他       : 无
10 论坛       : www.openedv.com
11 日志       : 初版 V1.0 2019/3/26 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 #include "bsp_gpio.h"
15 
16 /* SPI 片选信号 */
17 #define ICM20608_CSN(n)    (n ? gpio_pinwrite(GPIO1, 20, 1) : 
                               gpio_pinwrite(GPIO1, 20, 0))
18 
19 #define ICM20608G_ID       0XAF    /* ID 值 */
```

```
20 #define ICM20608D_ID      0XAE    /* ID 值 */
21 
22 /* ICM20608 寄存器
23  *复位后所有寄存器地址都为 0，除了
24  *Register 107(0X6B) Power Management 1    = 0x40
25  *Register 117(0X75) WHO_AM_I              = 0xAF 或者 0xAE
26  */
27 /* 陀螺仪和加速度自测(出产时设置，用于与用户的自检输出值比较） */
28 #define  ICM20_SELF_TEST_X_GYRO      0x00
29 #define  ICM20_SELF_TEST_Y_GYRO      0x01
30 #define  ICM20_SELF_TEST_Z_GYRO      0x02
31 #define  ICM20_SELF_TEST_X_ACCEL     0x0D
32 #define  ICM20_SELF_TEST_Y_ACCEL     0x0E
33 #define  ICM20_SELF_TEST_Z_ACCEL     0x0F
34 /***********省略掉其他宏定义************/
35 #define  ICM20_ZA_OFFSET_H           0x7D
36 #define  ICM20_ZA_OFFSET_L           0x7E
37 
38 /*
39  * ICM20608 结构体
40  */
41 struct icm20608_dev_struc
42 {
43     signed int gyro_x_adc;      /* 陀螺仪 X 轴原始值      */
44     signed int gyro_y_adc;      /* 陀螺仪 Y 轴原始值      */
45     signed int gyro_z_adc;      /* 陀螺仪 Z 轴原始值      */
46     signed int accel_x_adc;     /* 加速度计 X 轴原始值    */
47     signed int accel_y_adc;     /* 加速度计 Y 轴原始值    */
48     signed int accel_z_adc;     /* 加速度计 Z 轴原始值    */
49     signed int temp_adc;        /* 温度原始值            */
50 
51     /* 下面是计算得到的实际值，扩大 100 倍 */
52     signed int gyro_x_act;      /* 陀螺仪 X 轴实际值      */
53     signed int gyro_y_act;      /* 陀螺仪 Y 轴实际值      */
54     signed int gyro_z_act;      /* 陀螺仪 Z 轴实际值      */
55     signed int accel_x_act;     /* 加速度计 X 轴实际值    */
56     signed int accel_y_act;     /* 加速度计 Y 轴实际值    */
57     signed int accel_z_act;     /* 加速度计 Z 轴实际值    */
58     signed int temp_act;        /* 温度实际值            */
59 };
60 
61 struct icm20608_dev_struc icm20608_dev;  /* icm20608 设备 */
62 
```

```
63 /* 函数声明 */
64 unsigned char icm20608_init(void);
65 void icm20608_write_reg(unsigned char reg, unsigned char value);
66 unsigned char icm20608_read_reg(unsigned char reg);
67 void icm20608_read_len(unsigned char reg, unsigned char *buf,
                          unsigned char len);
68 void icm20608_getdata(void);
69 #endif
```

文件 bsp_icm20608.h 里面先定义了一个宏 ICM20608_CSN，这个是 ICM20608 的 SPI 片选引脚。接下来定义了一些 ICM20608 的 ID 和寄存器地址。第 41 行定义了一个结构体 icm20608_dev_struc，这个结构体是 ICM20608 的设备结构体，里面的成员变量用来保存 ICM20608 的原始数据值和经过转换得到的实际值。实际值是有小数的，本章例程取两位小数，为了方便计算，实际值扩大了 100 倍，这样实际值就是整数了，但是在使用的时候要除 100 重新得到小数部分。最后就是一些函数声明，接下来在文件 bsp_icm20608.c 中输入如下所示内容：

示例代码 27.3.4 bsp_icm20608.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名	: bsp_icm20608.c
作者	: 左忠凯
版本	: V1.0
描述	: ICM20608 驱动文件。
其他	: 无
论坛	: www.openedv.com
日志	: 初版 V1.0 2019/3/26 左忠凯创建
***************************************************************/
1  #include "bsp_icm20608.h"
2  #include "bsp_delay.h"
3  #include "bsp_spi.h"
4  #include "stdio.h"
5  
6  struct icm20608_dev_struc icm20608_dev; /* icm20608 设备 */
7  
8  /*
9   * @description  : 初始化 ICM20608
10  * @param        : 无
11  * @return       : 0 初始化成功，其他值 初始化失败
12  */
13 unsigned char icm20608_init(void)
14 {
15     unsigned char regvalue;
16     gpio_pin_config_t cs_config;
17 
18     /* 1、ESPI3 IO 初始化
```

```
19       * ECSPI3_SCLK  -> UART2_RXD
20       * ECSPI3_MISO  -> UART2_RTS
21       * ECSPI3_MOSI  -> UART2_CTS
22       */
23      IOMUXC_SetPinMux(IOMUXC_UART2_RX_DATA_ECSPI3_SCLK, 0);
24      IOMUXC_SetPinMux(IOMUXC_UART2_CTS_B_ECSPI3_MOSI, 0);
25      IOMUXC_SetPinMux(IOMUXC_UART2_RTS_B_ECSPI3_MISO, 0);
26      IOMUXC_SetPinConfig(IOMUXC_UART2_RX_DATA_ECSPI3_SCLK, 0x10B1);
27      IOMUXC_SetPinConfig(IOMUXC_UART2_CTS_B_ECSPI3_MOSI, 0x10B1);
28      IOMUXC_SetPinConfig(IOMUXC_UART2_RTS_B_ECSPI3_MISO, 0x10B1);
29  
30      /*  初始化片选引脚  */
31      IOMUXC_SetPinMux(IOMUXC_UART2_TX_DATA_GPIO1_IO20, 0);
32      IOMUXC_SetPinConfig(IOMUXC_UART2_TX_DATA_GPIO1_IO20, 0X10B0);
33      cs_config.direction = kGPIO_DigitalOutput;
34      cs_config.outputLogic = 0;
35      gpio_init(GPIO1, 20, &cs_config);
36  
37      /* 2、初始化 SPI */
38      spi_init(ECSPI3);
39  
40      icm20608_write_reg(ICM20_PWR_MGMT_1, 0x80);  /* 复位      */
41      delayms(50);
42      icm20608_write_reg(ICM20_PWR_MGMT_1, 0x01);  /* 关闭睡眠  */
43      delayms(50);
44  
45      regvalue = icm20608_read_reg(ICM20_WHO_AM_I);
46      printf("icm20608 id = %#X\r\n", regvalue);
47      if(regvalue != ICM20608G_ID && regvalue != ICM20608D_ID)
48          return 1;
49  
50      icm20608_write_reg(ICM20_SMPLRT_DIV, 0x00);   /* 输出速率设置     */
51      icm20608_write_reg(ICM20_GYRO_CONFIG, 0x18);  /* 陀螺仪±2000dps */
52      icm20608_write_reg(ICM20_ACCEL_CONFIG, 0x18); /* 加速度计±16G   */
53      icm20608_write_reg(ICM20_CONFIG, 0x04);       /* 陀螺 BW=20Hz   */
54      icm20608_write_reg(ICM20_ACCEL_CONFIG2, 0x04);
55      icm20608_write_reg(ICM20_PWR_MGMT_2, 0x00);   /* 打开所有轴       */
56      icm20608_write_reg(ICM20_LP_MODE_CFG, 0x00);  /* 关闭低功耗       */
57      icm20608_write_reg(ICM20_FIFO_EN, 0x00);      /* 关闭 FIFO      */
58      return 0;
59  }
60  
61  /*
```

```
62   * @description   : 写 ICM20608 指定寄存器
63   * @param - reg   : 要读取的寄存器地址
64   * @param – value : 要写入的值
65   * @return        : 无
66   */
67  void icm20608_write_reg(unsigned char reg, unsigned char value)
68  {
69      /* ICM20608 在使用 SPI 接口的时候寄存器地址只有低 7 位有效,
70       * 寄存器地址最高位是读/写标志位，读的时候要为 1，写的时候要为 0。
71       */
72      reg &= ~0X80;
73  
74      ICM20608_CSN(0);                        /* 使能 SPI 传输    */
75      spich0_readwrite_byte(ECSPI3, reg);     /* 发送寄存器地址   */
76      spich0_readwrite_byte(ECSPI3, value);   /* 发送要写入的值   */
77      ICM20608_CSN(1);                        /* 禁止 SPI 传输    */
78  }
79  
80  /*
81   * @description   : 读取 ICM20608 寄存器值
82   * @param - reg   : 要读取的寄存器地址
83   * @return        : 读取到的寄存器值
84   */
85  unsigned char icm20608_read_reg(unsigned char reg)
86  {
87      unsigned char reg_val;
88  
89      /* ICM20608 在使用 SPI 接口的时候寄存器地址只有低 7 位有效,
90       * 寄存器地址最高位是读/写标志位，读的时候要为 1，写的时候要为 0。
91       */
92      reg |= 0x80;
93  
94      ICM20608_CSN(0);                        /* 使能 SPI 传输          */
95      spich0_readwrite_byte(ECSPI3, reg);     /* 发送寄存器地址        */
96      reg_val = spich0_readwrite_byte(ECSPI3, 0XFF);/* 读取寄存器的值*/
97      ICM20608_CSN(1);                        /* 禁止 SPI 传输          */
98      return(reg_val);                        /* 返回读取到的寄存器值  */
99  }
100 
101 /*
102  * @description   : 读取 ICM20608 连续多个寄存器
103  * @param - reg   : 要读取的寄存器地址
104  * @return        : 读取到的寄存器值
```

```
105  */
106 void icm20608_read_len(unsigned char reg, unsigned char *buf,
                           unsigned char len)
107 {
108     unsigned char i;
109 
110     /* ICM20608 在使用 SPI 接口的时候寄存器地址，只有低 7 位有效,
111      * 寄存器地址最高位是读/写标志位读的时候要为 1，写的时候要为 0。
112      */
113     reg |= 0x80;
114 
115     ICM20608_CSN(0);                    /* 使能 SPI 传输        */
116     spich0_readwrite_byte(ECSPI3, reg);/* 发送寄存器地址        */
117     for(i = 0; i < len; i++)           /* 顺序读取寄存器的值    */
118     {
119         buf[i] = spich0_readwrite_byte(ECSPI3, 0XFF);
120     }
121     ICM20608_CSN(1);                    /* 禁止 SPI 传输        */
122 }
123 
124 /*
125  * @description   : 获取陀螺仪的分辨率
126  * @param         : 无
127  * @return        : 获取到的分辨率
128  */
129 float icm20608_gyro_scaleget(void)
130 {
131     unsigned char data;
132     float gyroscale;
133 
134     data = (icm20608_read_reg(ICM20_GYRO_CONFIG) >> 3) & 0X3;
135     switch(data) {
136         case 0:
137             gyroscale = 131;
138             break;
139         case 1:
140             gyroscale = 65.5;
141             break;
142         case 2:
143             gyroscale = 32.8;
144             break;
145         case 3:
146             gyroscale = 16.4;
```

```
147             break;
148     }
149     return gyroscale;
150 }
151 
152 /*
153  * @description   : 获取加速度计的分辨率
154  * @param         : 无
155  * @return        : 获取到的分辨率
156  */
157 unsigned short icm20608_accel_scaleget(void)
158 {
159     unsigned char data;
160     unsigned short accelscale;
161 
162     data = (icm20608_read_reg(ICM20_ACCEL_CONFIG) >> 3) & 0X3;
163     switch(data) {
164         case 0: 
165             accelscale = 16384;
166             break;
167         case 1: 
168             accelscale = 8192;
169             break;
170         case 2: 
171             accelscale = 4096;
172             break;
173         case 3: 
174             accelscale = 2048;
175             break;
176     }
177     return accelscale;
178 }
179 
180 /*
181  * @description   : 读取 ICM20608 的加速度、陀螺仪和温度原始值
182  * @param         : 无
183  * @return        : 无
184  */
185 void icm20608_getdata(void)
186 {
187     float gyroscale;
188     unsigned short accescale;
189     unsigned char data[14];
```

```
190
191     icm20608_read_len(ICM20_ACCEL_XOUT_H, data, 14);
192
193     gyroscale = icm20608_gyro_scaleget();
194     accescale = icm20608_accel_scaleget();
195
196     icm20608_dev.accel_x_adc = (signed short)((data[0] << 8) |
                                  data[1]);
197     icm20608_dev.accel_y_adc = (signed short)((data[2] << 8) |
                                  data[3]);
198     icm20608_dev.accel_z_adc = (signed short)((data[4] << 8) |
                                  data[5]);
199     icm20608_dev.temp_adc    = (signed short)((data[6] << 8) |
                                  data[7]);
200     icm20608_dev.gyro_x_adc  = (signed short)((data[8] << 8) |
                                  data[9]);
201     icm20608_dev.gyro_y_adc  = (signed short)((data[10] << 8) |
                                  data[11]);
202     icm20608_dev.gyro_z_adc  = (signed short)((data[12] << 8) |
                                  data[13]);
203
204     /* 计算实际值 */
205     icm20608_dev.gyro_x_act = ((float)(icm20608_dev.gyro_x_adc)  /
                                  gyroscale) * 100;
206     icm20608_dev.gyro_y_act = ((float)(icm20608_dev.gyro_y_adc)  /
                                  gyroscale) * 100;
207     icm20608_dev.gyro_z_act = ((float)(icm20608_dev.gyro_z_adc)  /
                                  gyroscale) * 100;
208     icm20608_dev.accel_x_act = ((float)(icm20608_dev.accel_x_adc) /
                                  accescale) * 100;
209     icm20608_dev.accel_y_act = ((float)(icm20608_dev.accel_y_adc) /
                                  accescale) * 100;
210     icm20608_dev.accel_z_act = ((float)(icm20608_dev.accel_z_adc) /
                                  accescale) * 100;
211     icm20608_dev.temp_act = (((float)(icm20608_dev.temp_adc) - 25) /
                                  326.8 + 25) * 100;
212 }
```

文件 bsp_imc20608.c 是 ICM20608 的驱动文件，里面有 7 个函数，我们依次来看一下。第 1 个函数是 icm20608_init，这个是 ICM20608 的初始化函数，此函数先初始化 ICM20608 所使用的 SPI 引脚，将其复用为 ECSPI3。因为我们本章的 SPI 片选采用软件控制的方式，所以 SPI 片选引脚设置成了普通的输出模式。设置完 SPI 所使用的引脚以后就是调用函数 spi_init 来初始化 SPI3，最后初始化 ICM20608，就是配置 ICM20608 的寄存器。第 2 个和第 3 个函数分别是 icm20608_write_reg 和 icm20608_read_reg，这两个函数分别用于写/读 ICM20608 的指定寄存

器。第 4 个函数是 icm20608_read_len，此函数也是读取 ICM20608 的寄存器值，但是此函数可以读取连续多个寄存器的值，一般用于读取 ICM20608 传感器数据。第 5 和第 6 个函数分别是 icm20608_gyro_scaleget 和 icm20608_accel_scaleget，这两个函数分别用于获取陀螺仪和加速度计的分辨率，因为陀螺仪和加速度的测量范围设置的不同，其分辨率就不同，所以在计算实际值的时候要根据实际的量程范围来得到对应的分辨率。最后一个函数是 icm20608_getdata，此函数就是用于获取 ICM20608 的加速度计、陀螺仪和温度计的数据，并且会根据设置的测量范围计算出实际的值，比如加速度的 g 值、陀螺仪的角速度值和温度计的温度值。

最后在 main.c 中输入如下内容：

示例代码 27.3.5 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 19 SPI 实验
其他      : SPI 也是最常用的接口，ALPHA 开发板上有一个 6 轴传感器 ICM20608，
            这个六轴传感器就是 SPI 接口的，本实验就来学习如何驱动 I.MX6U
            的 SPI 接口，并且通过 SPI 接口读取 ICM20608 的数据值。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/17 左忠凯创建
***************************************************************/
1   #include "bsp_clk.h"
2   #include "bsp_delay.h"
3   #include "bsp_led.h"
4   #include "bsp_beep.h"
5   #include "bsp_key.h"
6   #include "bsp_int.h"
7   #include "bsp_uart.h"
8   #include "bsp_lcd.h"
9   #include "bsp_rtc.h"
10  #include "bsp_icm20608.h"
11  #include "bsp_spi.h"
12  #include "stdio.h"
13
14  /*
15   * @description   : 指定的位置显示整数数据
16   * @param - x     : X 轴位置
17   * @param - y     : Y 轴位置
18   * @param - size  : 字体大小
19   * @param - num   : 要显示的数据
20   * @return        : 无
21   */
22  void integer_display(unsigned short x, unsigned short y,
```

```
                              unsigned char size, signed int num)
23 {
24     char buf[200];
25 
26     lcd_fill(x, y, x + 50, y + size, tftlcd_dev.backcolor);
27 
28     memset(buf, 0, sizeof(buf));
29     if(num < 0)
30         sprintf(buf, "-%d", -num);
31     else
32         sprintf(buf, "%d", num);
33     lcd_show_string(x, y, 50, size, size, buf);
34 }
35 
36 /*
37  * @description  : 指定的位置显示小数数据,比如 5123，显示为 51.23
38  * @param - x     : X 轴位置
39  * @param - y     : Y 轴位置
40  * @param - size : 字体大小
41  * @param - num  : 要显示的数据，实际小数扩大 100 倍，
42  * @return       : 无
43  */
44 void decimals_display(unsigned short x, unsigned short y,
                       unsigned char size, signed int num)
45 {
46     signed int integ;   /* 整数部分 */
47     signed int fract;   /* 小数部分 */
48     signed int uncomptemp = num;
49     char buf[200];
50 
51     if(num < 0)
52         uncomptemp = -uncomptemp;
53     integ = uncomptemp / 100;
54     fract = uncomptemp % 100;
55 
56     memset(buf, 0, sizeof(buf));
57     if(num < 0)
58         sprintf(buf, "-%d.%d", integ, fract);
59     else
60         sprintf(buf, "%d.%d", integ, fract);
61     lcd_fill(x, y, x + 60, y + size, tftlcd_dev.backcolor);
62     lcd_show_string(x, y, 60, size, size, buf);
63 }
```

```
64
65  /*
66   * @description   : 使能 I.MX6U 的硬件 NEON 和 FPU
67   * @param         : 无
68   * @return        : 无
69   */
70   void imx6ul_hardfpu_enable(void)
71  {
72      uint32_t cpacr;
73      uint32_t fpexc;
74
75      /* 使能 NEON 和 FPU */
76      cpacr = __get_CPACR();
77      cpacr = (cpacr & ~(CPACR_ASEDIS_Msk | CPACR_D32DIS_Msk))
78             |  (3UL << CPACR_cp10_Pos) | (3UL << CPACR_cp11_Pos);
79      __set_CPACR(cpacr);
80      fpexc = __get_FPEXC();
81      fpexc |= 0x40000000UL;
82      __set_FPEXC(fpexc);
83  }
84
85  /*
86   * @description : main 函数
87   * @param       : 无
88   * @return      : 无
89   */
90  int main(void)
91  {
92      unsigned char state = OFF;
93
94      imx6ul_hardfpu_enable();    /* 使能 I.MX6U 的硬件浮点        */
95      int_init();                 /* 初始化中断(一定要最先调用！) */
96      imx6u_clkinit();            /* 初始化系统时钟                */
97      delay_init();               /* 初始化延时                    */
98      clk_enable();               /* 使能所有的时钟                */
99      led_init();                 /* 初始化 led                    */
100     beep_init();                /* 初始化 beep                   */
101     uart_init();                /* 初始化串口，波特率 115200     */
102     lcd_init();                 /* 初始化 LCD                    */
103
104     tftlcd_dev.forecolor = LCD_RED;
105     lcd_show_string(50, 10, 400, 24, 24,
                        (char*)"IMX6U-ALPHA SPI TEST");
```

```
106     lcd_show_string(50, 40, 200, 16, 16, (char*)"ICM20608 TEST");
107     lcd_show_string(50, 60, 200, 16, 16, (char*)"ATOM@ALIENTEK");
108     lcd_show_string(50, 80, 200, 16, 16, (char*)"2019/3/27");
109 
110     while(icm20608_init())      /* 初始化 ICM20608 */
111     {
112         lcd_show_string(50, 100, 200, 16, 16,
                        (char*)"ICM20608 Check Failed!");
113         delayms(500);
114         lcd_show_string(50, 100, 200, 16, 16,
                        (char*)"Please Check!       ");
115         delayms(500);
116     }
117     lcd_show_string(50, 100, 200, 16, 16, (char*)"ICM20608 Ready");
118     lcd_show_string(50, 130, 200, 16, 16, (char*)"accel x:");
119     lcd_show_string(50, 150, 200, 16, 16, (char*)"accel y:");
120     lcd_show_string(50, 170, 200, 16, 16, (char*)"accel z:");
121     lcd_show_string(50, 190, 200, 16, 16, (char*)"gyro  x:");
122     lcd_show_string(50, 210, 200, 16, 16, (char*)"gyro  y:");
123     lcd_show_string(50, 230, 200, 16, 16, (char*)"gyro  z:");
124     lcd_show_string(50, 250, 200, 16, 16, (char*)"temp   :");
125     lcd_show_string(50 + 181, 130, 200, 16, 16, (char*)"g");
126     lcd_show_string(50 + 181, 150, 200, 16, 16, (char*)"g");
127     lcd_show_string(50 + 181, 170, 200, 16, 16, (char*)"g");
128     lcd_show_string(50 + 181, 190, 200, 16, 16, (char*)"o/s");
129     lcd_show_string(50 + 181, 210, 200, 16, 16, (char*)"o/s");
130     lcd_show_string(50 + 181, 230, 200, 16, 16, (char*)"o/s");
131     lcd_show_string(50 + 181, 250, 200, 16, 16, (char*)"C");
132 
133     tftlcd_dev.forecolor = LCD_BLUE;
134 
135     while(1)
136     {
137         icm20608_getdata();     /* 获取数据值 */
138         /* 在 LCD 上显示原始值 */
139         integer_display(50 + 70, 130, 16, icm20608_dev.accel_x_adc);
140         integer_display(50 + 70, 150, 16, icm20608_dev.accel_y_adc);
141         integer_display(50 + 70, 170, 16, icm20608_dev.accel_z_adc);
142         integer_display(50 + 70, 190, 16, icm20608_dev.gyro_x_adc);
143         integer_display(50 + 70, 210, 16, icm20608_dev.gyro_y_adc);
144         integer_display(50 + 70, 230, 16, icm20608_dev.gyro_z_adc);
145         integer_display(50 + 70, 250, 16, icm20608_dev.temp_adc);
146 
```

```
147         /* 在 LCD 上显示计算得到的原始值 */
148         decimals_display(50 + 70 + 50, 130, 16,
                                icm20608_dev.accel_x_act);
149         decimals_display(50 + 70 + 50, 150, 16,
                                icm20608_dev.accel_y_act);
150         decimals_display(50 + 70 + 50, 170, 16,
                                icm20608_dev.accel_z_act);
151         decimals_display(50 + 70 + 50, 190, 16,
                                icm20608_dev.gyro_x_act);
152         decimals_display(50 + 70 + 50, 210, 16,
                                icm20608_dev.gyro_y_act);
153         decimals_display(50 + 70 + 50, 230, 16,
                                icm20608_dev.gyro_z_act);
154         decimals_display(50 + 70 + 50, 250, 16,
                                icm20608_dev.temp_act);
155         delayms(120);
156         state = !state;
157         led_switch(LED0,state);
158     }
159     return 0;
160 }
```

文件 main.c 一开始有两个函数 integer_display 和 decimals_display，这两个函数用于在 LCD 上显示获取到的 ICM20608 数据值，函数 integer_display 用于显示原始数据值，也就是整数值。函数 decimals_display 用于显示实际值，实际值扩大了 100 倍，此函数会提取出实际值的整数部分和小数部分并显示在 LCD 上。另一个重要的函数是 imx6ul_hardfpu_enable，这个函数用于开启 I.MX6U 的 NEON 和硬件 FPU(浮点运算单元)，因为本章使用到了浮点运算，而 I.MX6U 的 Cortex-A7 是支持 NEON 和 FPU(VFPV4_D32)的，但是在使用 I.MX6U 的硬件 FPU 之前是先要开启的。

第 110 行调用了函数 icm20608_init 来初始化 ICM20608，如果初始化失败的话就会在 LCD 上闪烁提示语句。最后在 main 函数的 while 循环中不断的调用函数 icm20608_getdata 获取 ICM20608 的传感器数据，并且显示在 LCD 上。实验程序编写就到这里结束了，接下来就是编译、下载和验证了。

27.4 编译下载验证

27.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 icm20608，然后在在 INCDIRS 和 SRCDIRS 中加入“bsp/spi”和“bsp/icm20608”，修改后的 Makefile 如下：

示例代码 27.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= icm20608
3
4  /* 省略掉其它代码...... */
```

```
5
6  INCDIRS        := imx6ul \
7                    stdio/include \
8                    bsp/clk \
9                    bsp/led \
10                   bsp/delay  \
11                   bsp/beep \
12                   bsp/gpio \
13                   bsp/key \
14                   bsp/exit \
15                   bsp/int \
16                   bsp/epittimer \
17                   bsp/keyfilter \
18                   bsp/uart \
19                   bsp/lcd \
20                   bsp/rtc \
21                   bsp/i2c \
22                   bsp/ap3216c \
23                   bsp/spi \
24                   bsp/icm20608
25
26 SRCDIRS        := project \
27                   stdio/lib \
28                   bsp/clk \
29                   bsp/led \
30                   bsp/delay \
31                   bsp/beep \
32                   bsp/gpio \
33                   bsp/key \
34                   bsp/exit \
35                   bsp/int \
36                   bsp/epittimer \
37                   bsp/keyfilter \
38                   bsp/uart \
39                   bsp/lcd \
40                   bsp/rtc \
41                   bsp/i2c \
42                   bsp/ap3216c \
43                   bsp/spi \
44                   bsp/icm20608
45
46 /* 省略掉其它代码...... */
47
```

```
48 $(COBJS) : obj/%.o : %.c
49  $(CC) -Wall -march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=hard -Wa,
          -mimplicit-it=thumb -nostdlib -fno-builtin
          -c -O2  $(INCLUDE) -o $@ $<
50
51 clean:
52  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“icm20608”，也就是目标名称为“ap3216c”。

第 23 和 24 行在变量 INCDIRS 中添加 SPI 和 ICM20608 的驱动头文件(.h)路径。

第 43 和 44 行在变量 SRCDIRS 中添加 SPI 和 ICM20608 驱动文件(.c)路径。

第 49 行加入了“-march=armv7-a -mfpu=neon-vfpv4 -mfloat-abi=hard”指令，这些指令用于指定编译浮点运算的时候使用硬件 FPU。因为本章使用到了浮点运算，而 I.MX6U 是支持硬件 FPU 的，虽然我们在 main 函数中已经打开了 NEON 和 FPU，但是在编译相应 C 文件的时候也要指定使用硬件 FPU 来编译浮点运算。

链接脚本保持不变。

27.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 icm20608.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限，一次即可
./imxdownload icm20608.bin /dev/sdd      //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。如果 ICM20608 工作正常的话就会在 LCD 上显示获取到的传感器数据，如图 27.4.2.1 所示：

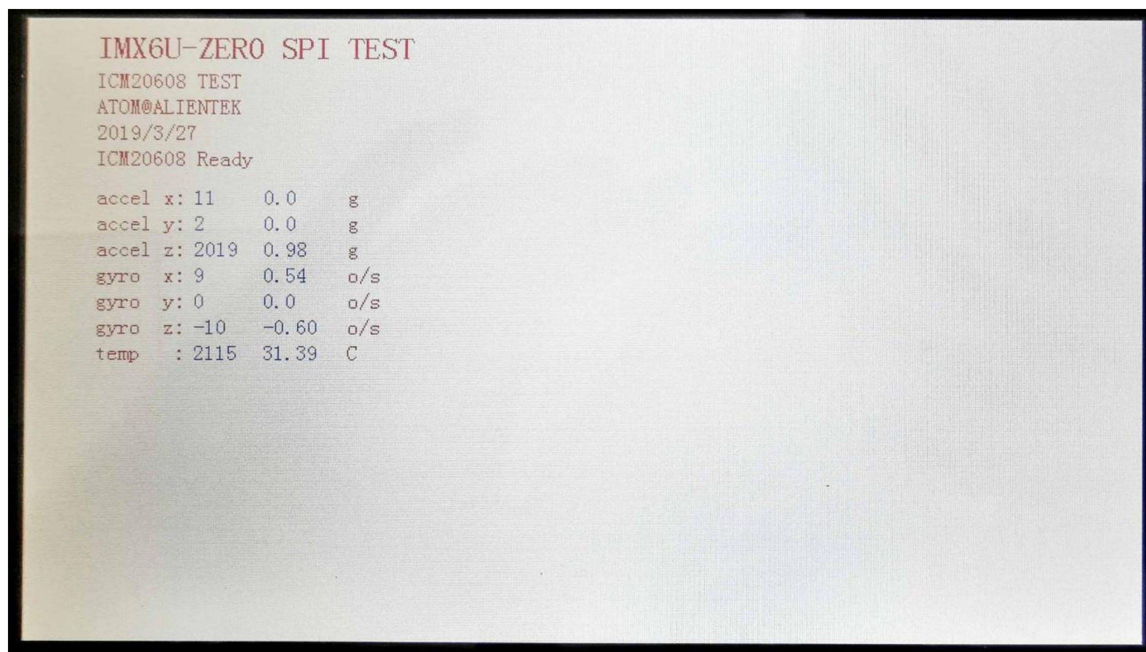


图 27.4.2.1 LCD 界面

在图 27.4.2.1 中可以看到加速度计 Z 轴在静止状态下是 0.98g，这不正是重力加速度。温度传感器测量到的温度是 31.39° C，这个是芯片内部的温度，并不是室温！芯片内部温度一般要比室温高。如果动一下开发板的话加速度计和陀螺仪的数据就会变化。

第二十八章 多点电容触摸屏实验

随着智能手机的发展，电容触摸屏也得到了飞速的发展。相比电阻触摸屏，电容触摸屏有很多的优势，比如支持多点触控、不需要按压，只需要轻轻触摸就有反应。ALIENTEK 的三款 RGB LCD 屏幕都支持多点电容触摸，本章就以 ATK7016 这款 RGB LCD 屏幕为例讲解一下如何驱动电容触摸屏，并获取对应的触摸坐标值。

28.1 多点电容触摸简介

触摸屏很早就有了，一开始是电阻触摸屏，电阻触摸屏只能单点触摸，在以前的学习机、功能机时代被广泛使用。2007 年 1 月 9 日苹果发布了划时代的第一代 Iphone，也就是 Iphone 2G，Iphone 2G 上使用了多点电容触摸屏，而当时的手机基本都是使用的电阻触摸屏。电容触摸屏优秀的触摸品质和手感瞬间征服了消费者，带来了手机触摸屏的大变革，后面新出的手机也都采用了多点电容触摸屏。和电阻触摸屏相比，电容触摸屏最大的优点是支持多点触摸(后面的电阻屏也支持多点触摸，但是为时已晚)，电容屏只需要手指轻触即可，而电阻屏是需要手指给予一定的压力才有反应，而且电容屏不需要校准。如今多点电容触摸屏已经得到了广泛的应用，手机、平板、电脑、广告机等等，如果要做人机交互设备的开发，多点电容触摸屏基本是不可能绕过去的。所以本章我们就来学习一下如何使用多点触摸屏，如何获取到多点触摸值。关于电容屏的物理原理我们就不去研究了，毕竟我们不是开发电容屏的，而是电容屏的使用者，我们只需要关注如何使用电容屏，如何得到其多点触摸坐标值即可。ALIENTEK 的三款 RGB LCD 屏幕都是支持 5 点电容触摸屏的，本章我们同样以 ATK-7016 这款屏幕为例来讲解如何使用多点电容触摸屏。

ATK-7016 这款屏幕其实是由 TFT LCD+触摸屏组合起来的。底下是 LCD 面板，上面是触摸面板，将两个封装到一起就成了带有触摸屏的 LCD 屏幕。电容触摸屏也是需要一个驱动 IC 的，驱动 IC 一般会提供一个 I2C 接口给主控制器，主控制器可以通过 I2C 接口来读取驱动 IC 里面的触摸坐标数据。ATK-7016、ATK-7084 这两款屏幕使用的触摸控制 IC 是 FT5426，ATK-4342 使用的驱动 IC 是 GT9147。这三个电容屏触摸 IC 都是 I2C 接口的，使用方法基本一样。

FT5426 这款驱动 IC 采用 15*28 的驱动结构，也就是 15 个感应通道，28 个驱动通道，最多支持 5 点电容触摸。ATK-7016 的电容触摸屏部分有 4 个 IO 用于连接主控制器：SCL、SDA、RST 和 INT，SCL 和 SDA 是 I2C 引脚，RST 是复位引脚，INT 是中断引脚。一般通过 INT 引脚来通知主控制器有触摸点按下，然后在 INT 中断服务函数中读取触摸数据。也可以不使用中断功能，采用轮询的方式不断查询是否有触摸点按下，本章实验我们使用中断方式来获取触摸数据。

根所有的 I2C 器件一样，FT5426 也是通过读写寄存器来完成初始化和触摸坐标数据读取的，I.MX6U 的 I2C 我们已经在第二十六章做了详细的讲解，所以本章的主要工作就是读写 FT5426 的寄存器。FT5426 的 I2C 设备地址为 0X38，FT5426 的寄存器有很多，本章我们只用到了其中的一部分，如表 28.1.1 所示：

寄存器地址	位	寄存器功能	描述
0X00	[6:4]	模式寄存器	设置 FT5426 的工作模式： 000：正常模式。 001：系统信息模式 100：测试模式。
0X02	[3:0]	触摸状态寄存器	记录有多少个触摸点， 有效值为 1~5。
0X03	[7:6]	第一个触摸点 X 坐标高位数据	事件标志： 00：按下。 01：抬起 10：接触 11：保留
	[3:0]		X 轴坐标值高 4 位。

0X04	[7:0]	第一个触摸点 X 坐标低位数据	X 轴坐标值低 8 位
0X05	[7:4]	第一个触摸点 Y 坐标高位数据	触摸点的 ID。
	[3:0]		Y 轴坐标高 4 位
0X06	[7:0]	第一个触摸点 Y 坐标低位数据	Y 轴坐标低 8 位
0X09	[7:6]	第二个触摸点 X 坐标高位数据	与寄存器 0X03 含义相同。
	[3:0]		
0X0A	[7:0]	第二个触摸点 X 坐标低位数据	与寄存器 0X04 含义相同。
0X0B	[7:4]	第二个触摸点 Y 坐标高位数据	与寄存器 0X05 含义相同。
	[3:0]		
0X0C	[7:0]	第二个触摸点 Y 坐标低位数据	与寄存器 0X06 含义相同
0X0F	[7:6]	第三个触摸点 X 坐标高位数据	与寄存器 0X03 含义相同。
	[3:0]		
0X10	[7:0]	第三个触摸点 X 坐标低位数据	与寄存器 0X04 含义相同。
0X11	[7:4]	第三个触摸点 Y 坐标高位数据	与寄存器 0X05 含义相同。
	[3:0]		
0X12	[7:0]	第三个触摸点 Y 坐标低位数据	与寄存器 0X06 含义相同
0X15	[7:6]	第四个触摸点 X 坐标高位数据	与寄存器 0X03 含义相同。
	[3:0]		
0X16	[7:0]	第四个触摸点 X 坐标低位数据	与寄存器 0X04 含义相同。
0X17	[7:4]	第四个触摸点 Y 坐标高位数据	与寄存器 0X05 含义相同。
	[3:0]		
0X18	[7:0]	第四个触摸点 Y 坐标低位数据	与寄存器 0X06 含义相同
0X1B	[7:6]	第五个触摸点 X 坐标高位数据	与寄存器 0X03 含义相同。
	[3:0]		
0X1C	[7:0]	第五个触摸点 X 坐标低位数据	与寄存器 0X04 含义相同。
0X1D	[7:4]	第五个触摸点 Y 坐标高位数据	与寄存器 0X05 含义相同。
	[3:0]		
0X1E	[7:0]	第五个触摸点 Y 坐标低位数据	与寄存器 0X06 含义相同
0XA1	[7:0]	版本寄存器	版本高字节
0XA2	[7:0]		版本低字节
0XA4	[7:0]	中断模式寄存器	用于设置中断模式： 0：轮询模式 1：触发模式

表 28.1.1.1 FT5426 使用到的寄存器表

表 28.1.1.1 中就是本章实验我们会使用到的寄存器。关于触摸屏和 FT5426 的知识就讲解到这里。

28.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、 RGB LCD 屏幕。

③、触摸屏

④、串口

触摸屏是和 RGB LCD 屏幕做在一起的，所以触摸屏也在 RGB LCD 接口上，都是连接在 I.MX6U-ALPHA 开发板底板上，原理图如图 28.2.1 所示：

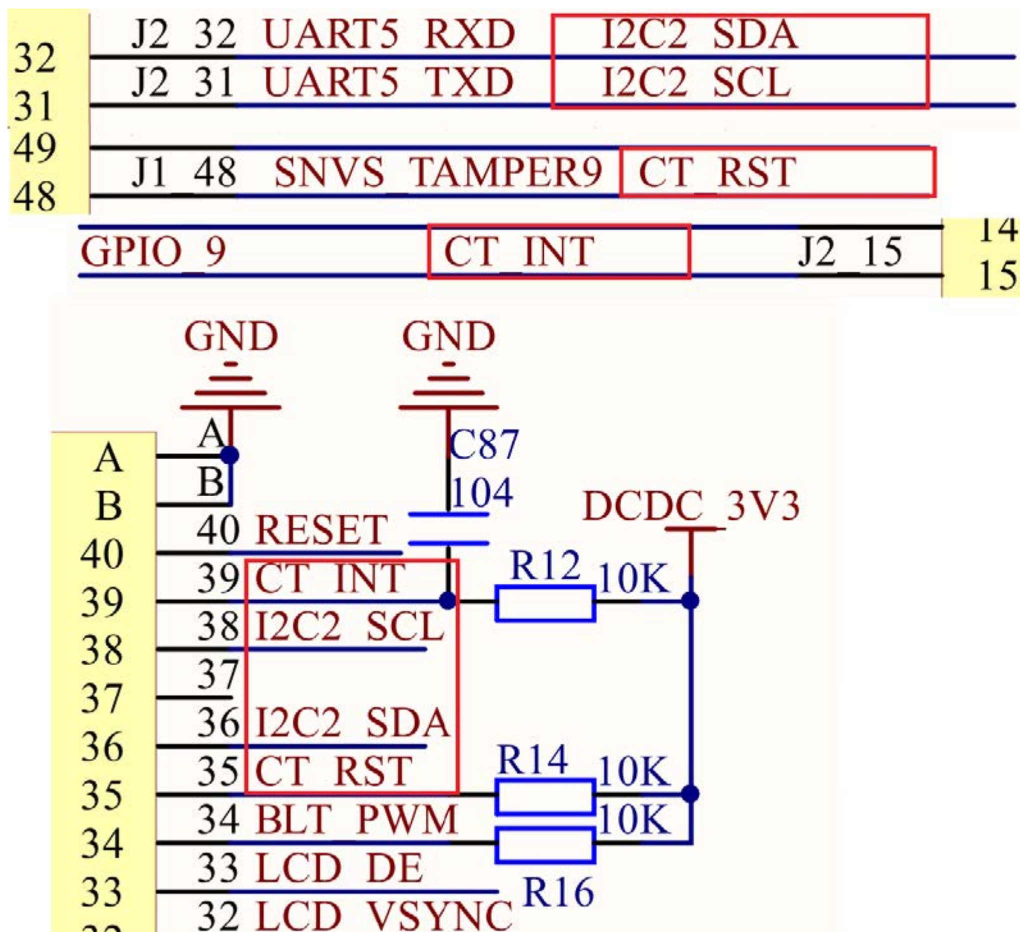


图 28.2.1 触摸屏原理图

从图 28.2.1 可以看出，触摸屏连接这 I.MX6U 的 I2C2，INT 引脚连接着 I.MX6U 的 SNVS_TAMPER9，RST 引脚连接着 I.MX6U 的 GPIO1_IO9。在本章实验中使用中断方式读取触摸点个数和触摸点坐标数据，并且将其显示在 LCD 上。

28.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 19_touchscreen。

本章实验在上一章例程的基础上完成，更改工程名字为“touchscreen”，然后在 bsp 文件夹下创建名为“touchscreen”的文件。在 bsp/ touchscreen 中新建 bsp_ft5xx6.c 和 bsp_ft5xx6.h 这两个文件，在 bsp_ft5xx6.h 中输入如下内容：

示例代码 28.3.1 bsp_ft5xx6.h 文件代码

```
1  #ifndef _FT5XX6_H
2  #define _FT5XX6_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名     : bsp_ft5xx6.h
6  作者       : 左忠凯
```

```
7  版本      : V1.0
8  描述      : 触摸屏驱动头文件,触摸芯片为 FT5xx6,
9              包括 FT5426 和 FT5406。
10 其他      : 无
11 论坛      : www.openedv.com
12 日志      : 初版 V1.0 2019/1/21 左忠凯创建
13 *****************************************************************/
14 #include "imx6ul.h"
15 #include "bsp_gpio.h"
16
17 /* 宏定义 */
18 #define FT5426_ADDR                0X38    /* FT5426 设备地址  */
19
20 #define FT5426_DEVICE_MODE         0X00    /* 模式寄存器        */
21 #define FT5426_IDGLIB_VERSION      0XA1    /* 固件版本寄存器    */
22 #define FT5426_IDG_MODE            0XA4    /* 中断模式          */
23 #define FT5426_TD_STATUS           0X02    /* 触摸状态寄存器    */
24 #define FT5426_TOUCH1_XH           0X03    /* 触摸点坐标寄存器,
25                                             * 一个触摸点用 4 个寄存器*/
26
27 #define FT5426_XYCOORDREG_NUM      30      /* 触摸点坐标寄存器数量  */
28 #define FT5426_INIT_FINISHED       1       /* 触摸屏初始化完成      */
29 #define FT5426_INIT_NOTFINISHED 0          /* 触摸屏初始化未完成    */
30
31 #define FT5426_TOUCH_EVENT_DOWN          0x00    /* 按下      */
32 #define FT5426_TOUCH_EVENT_UP            0x01    /* 释放      */
33 #define FT5426_TOUCH_EVENT_ON            0x02    /* 接触      */
34 #define FT5426_TOUCH_EVENT_RESERVED      0x03    /* 没有事件  */
35
36 /* 触摸屏结构体 */
37 struct ft5426_dev_struc
38 {
39  unsigned char initfalg;      /* 触摸屏初始化状态      */
40  unsigned char intflag;       /* 标记中断有没有发生    */
41  unsigned char point_num;     /* 触摸点                */
42  unsigned short x[5];         /* X 轴坐标              */
43  unsigned short y[5];         /* Y 轴坐标              */
44 };
45
46 extern struct ft5426_dev_struc ft5426_dev;
47
48 /* 函数声明 */
49 void ft5426_init(void);
```

```
50
51 void gpio1_io9_irqhandler(void);
52 unsigned char ft5426_write_byte(unsigned char addr,
                                   unsigned char reg,
                                   unsigned char data);
53 unsigned char ft5426_read_byte(unsigned char addr,
                                  unsigned char reg);
54 void ft5426_read_len(unsigned char addr,unsigned char reg,
                        unsigned char len,unsigned char *buf);
55 void ft5426_read_tpnum(void);
56 void ft5426_read_tpcoord(void);
57 #endif
```

文件 bsp_ft5xx6.h 文件中先是定义了 FT5426 的设备地址、寄存器地址和一些触摸点状态宏，然后在第 37 行定义了一个结构体 ft5426_dev_struc，此结构体用来保存触摸信息，最后就是一些函数声明。接下来在 bsp_ft5xx6.c 中输入如下所示内容：

示例代码 28.3.2 bsp_ft5xx6.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名      :  bsp_ft5xx6.c
作者        : 左忠凯
版本        : V1.0
描述        : 触摸屏驱动文件,触摸芯片为 FT5xx6,
              包括 FT5426 和 FT5406。
其他        : 无
论坛        : www.openedv.com
日志        : 初版 V1.0 2019/1/21 左忠凯创建
***************************************************************/
1   #include "bsp_ft5xx6.h"
2   #include "bsp_i2c.h"
3   #include "bsp_int.h"
4   #include "bsp_delay.h"
5   #include "stdio.h"
6
7   struct ft5426_dev_struc ft5426_dev;
8
9   /*
10   * @description   : 初始化触摸屏，其实就是初始化 FT5426
11   * @param         : 无
12   * @return        : 无
13   */
14  void ft5426_init(void)
15  {
16      unsigned char reg_value[2];
```

```
17
18      ft5426_dev.initfalg = FT5426_INIT_NOTFINISHED;
19
20      /* 1、初始化 IIC2 IO
21       * I2C2_SCL -> UART5_TXD
22       * I2C2_SDA -> UART5_RXD
23       */
24      IOMUXC_SetPinMux(IOMUXC_UART5_TX_DATA_I2C2_SCL, 1);
25      IOMUXC_SetPinMux(IOMUXC_UART5_RX_DATA_I2C2_SDA, 1);
26      IOMUXC_SetPinConfig(IOMUXC_UART5_TX_DATA_I2C2_SCL, 0x70B0);
27      IOMUXC_SetPinConfig(IOMUXC_UART5_RX_DATA_I2C2_SDA, 0X70B0);
28
29      /* 2、初始化触摸屏中断 IO 和复位 IO */
30      gpio_pin_config_t ctintpin_config;
31      IOMUXC_SetPinMux(IOMUXC_GPIO1_IO09_GPIO1_IO09,0);
32      IOMUXC_SetPinMux(IOMUXC_SNVS_SNVS_TAMPER9_GPIO5_IO09,0);
33      IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO09_GPIO1_IO09,0xF080);
34      IOMUXC_SetPinConfig(IOMUXC_SNVS_SNVS_TAMPER9_GPIO5_IO09,0X10B0);
35
36      /* 中断 IO 初始化 */
37      ctintpin_config.direction = kGPIO_DigitalInput;
38      ctintpin_config.interruptMode = kGPIO_IntRisingOrFallingEdge;
39      gpio_init(GPIO1, 9, &ctintpin_config);
40
41      GIC_EnableIRQ(GPIO1_Combined_0_15_IRQn); /* 使能 GIC 中对应的中断 */
42      system_register_irqhandler(GPIO1_Combined_0_15_IRQn,
                                  (system_irq_handler_t)gpio1_io9_irqhandler,
                                   NULL); /* 注册中断服务函数 */
43      gpio_enableint(GPIO1, 9);  /* 使能 GPIO1_IO18 的中断功能 */
44
45      /* 复位 IO 初始化 */
46      ctintpin_config.direction=kGPIO_DigitalOutput;
47      ctintpin_config.interruptMode=kGPIO_NoIntmode;
48      ctintpin_config.outputLogic=1;
49      gpio_init(GPIO5, 9, &ctintpin_config);
50
51      /* 3、初始化 I2C */
52      i2c_init(I2C2);
53
54      /* 4、初始化 FT5426 */
55      gpio_pinwrite(GPIO5, 9, 0); /* 复位 FT5426     */
56      delayms(20);
57      gpio_pinwrite(GPIO5, 9, 1); /* 停止复位 FT5426  */
```

```
58     delayms(20);
59     ft5426_write_byte(FT5426_ADDR, FT5426_DEVICE_MODE, 0);
60     ft5426_write_byte(FT5426_ADDR, FT5426_IDG_MODE, 1);
61     ft5426_read_len(FT5426_ADDR, FT5426_IDGLIB_VERSION, 2,
                       reg_value);
62     printf("Touch Frimware Version:%#X\r\n",
              ((unsigned short)reg_value[0] << 8) + reg_value[1]);
63     ft5426_dev.initfalg = FT5426_INIT_FINISHED; /* 标记初始化完成 */
64     ft5426_dev.intflag = 0;
65 }
66 
67 /*
68  * @description    : GPIO1_IO9 最终的中断处理函数
69  * @param          : 无
70  * @return         : 无
71  */
72 void gpio1_io9_irqhandler(void)
73 {
74     if(ft5426_dev.initfalg == FT5426_INIT_FINISHED)
75     {
76         //ft5426_dev.intflag = 1;
77         ft5426_read_tpcoord();
78     }
79     gpio_clearintflags(GPIO1, 9); /* 清除中断标志位 */
80 }
81 
82 /*
83  * @description    : 向 FT5426 写入数据
84  * @param - addr   : 设备地址
85  * @param - reg    : 要写入的寄存器
86  * @param - data   : 要写入的数据
87  * @return         : 操作结果
88  */
89 unsigned char ft5426_write_byte(unsigned char addr,
                                  unsigned char reg,
                                  unsigned char data)
90 {
91     unsigned char status=0;
92     unsigned char writedata=data;
93     struct i2c_transfer masterXfer;
94 
95     /* 配置 I2C xfer 结构体 */
96     masterXfer.slaveAddress = addr;        /* 设备地址              */
```

```
97      masterXfer.direction = kI2C_Write;      /* 写入数据           */
98      masterXfer.subaddress = reg;            /* 要写入的寄存器地址   */
99      masterXfer.subaddressSize = 1;          /* 地址长度一个字节     */
100     masterXfer.data = &writedata;           /* 要写入的数据         */
101     masterXfer.dataSize = 1;                /* 写入数据长度 1 个字节 */
102 
103     if(i2c_master_transfer(I2C2, &masterXfer))
104         status=1;
105 
106     return status;
107 }
108 
109 /*
110  * @description   : 从 FT5426 读取一个字节的数据
111  * @param - addr  : 设备地址
112  * @param - reg   : 要读取的寄存器
113  * @return        : 读取到的数据。
114  */
115 unsigned char ft5426_read_byte(unsigned char addr,
                                    unsigned char reg)
116 {
117     unsigned char val=0;
118 
119     struct i2c_transfer masterXfer;
120     masterXfer.slaveAddress = addr;         /* 设备地址           */
121     masterXfer.direction = kI2C_Read;       /* 读取数据           */
122     masterXfer.subaddress = reg;            /* 要读取的寄存器地址   */
123     masterXfer.subaddressSize = 1;          /* 地址长度一个字节     */
124     masterXfer.data = &val;                 /* 接收数据缓冲区       */
125     masterXfer.dataSize = 1;                /* 读取数据长度 1 个字节 */
126     i2c_master_transfer(I2C2, &masterXfer);
127     return val;
128 }
129 
130 /*
131  * @description   : 从 FT5429 读取多个字节的数据
132  * @param - addr  : 设备地址
133  * @param - reg   : 要读取的开始寄存器地址
134  * @param - len   : 要读取的数据长度.
135  * @param - buf   : 读取到的数据缓冲区
136  * @return        : 无
137  */
138 void ft426_read_len(unsigned char addr,unsigned char reg,
```

```
                         unsigned char len,unsigned char *buf)
139 {
140     struct i2c_transfer masterXfer;
141 
142     masterXfer.slaveAddress = addr;        /* 设备地址           */
143     masterXfer.direction = kI2C_Read;      /* 读取数据           */
144     masterXfer.subaddress = reg;           /* 要读取的寄存器地址   */
145     masterXfer.subaddressSize = 1;         /* 地址长度一个字节     */
146     masterXfer.data = buf;                 /* 接收数据缓冲区       */
147     masterXfer.dataSize = len;             /* 读取数据长度 1 个字节 */
148     i2c_master_transfer(I2C2, &masterXfer);
149 }
150 
151 /*
152  * @description   : 读取当前触摸点个数
153  * @param         : 无
154  * @return        : 无
155  */
156 void ft5426_read_tpnum(void)
157 {
158     ft5426_dev.point_num = ft5426_read_byte(FT5426_ADDR,
                                        FT5426_TD_STATUS);
159 }
160 
161 /*
162  * @description   : 读取当前所有触摸点的坐标
163  * @param         : 无
164  * @return        : 无
165  */
166 void ft5426_read_tpcoord(void)
167 {
168     unsigned char i = 0;
169     unsigned char type = 0;
170     //unsigned char id = 0;
171     unsigned char pointbuf[FT5426_XYCOORDREG_NUM];
172 
173     ft5426_dev.point_num = ft5426_read_byte(FT5426_ADDR,
                                        FT5426_TD_STATUS);
174 
175     /*
176      * 从寄存器 FT5426_TOUCH1_XH 开始，连续读取 30 个寄存器的值，
177      * 这 30 个寄存器保存着 5 个点的触摸值，每个点占用 6 个寄存器。
178      */
```

```
179     ft5426_read_len(FT5426_ADDR, FT5426_TOUCH1_XH,
                        FT5426_XYCOORDREG_NUM, pointbuf);
180     for(i = 0; i < ft5426_dev.point_num ; i++)
181     {
182         unsigned char *buf = &pointbuf[i * 6];

183         ft5426_dev.x[i] = ((buf[2] << 8) | buf[3]) & 0x0fff;
184         ft5426_dev.y[i] = ((buf[0] << 8) | buf[1]) & 0x0fff;
185         type = buf[0] >> 6; /* 获取触摸类型 */
186         //id = (buf[2] >> 4) & 0x0f;
187         if(type == FT5426_TOUCH_EVENT_DOWN || type ==
                        FT5426_TOUCH_EVENT_ON )/* 按下  */
188         {
189 
190         } else {   /* 释放 */
191 
192         }
193     }
194 }
```

文件 bsp_ft5xx6.c 中有 7 个函数，我们依次来看一下这 7 个函数。第 1 个是函数 ft5426_init，此函数是 ft5426 的初始化函数，此函数先初始化 FT5426 所使用的 I2C2 接口引脚、复位引脚和中断引脚。接下来使能了 FT5426 所使用的中断，并且注册了中断处理函数，最后初始化了 I2C2 和 FT5426。第 2 个函数是 gpio1_io9_irqhandler，这个是 FT5426 的中断引脚中断处理函数，在此函数中会读取 FT5426 内部的触摸数据。第 3 和第 4 个函数分别为 ft5426_write_byte 和 ft5426_read_byte，函数 ft5426_write_byte 用于向 FT5426 的寄存器写入指定的值，函数 ft5426_read_byte 用于读取 FT5426 指定寄存器的值。第 5 个函数是 ft5426_read_len，此函数也是从 FT5426 的指定寄存器读取数据，但是此函数是读取数个连续的寄存器。第 6 个函数是 ft5426_read_tpnum，此函数用于获取 FT5426 当前有几个触摸点有效，也就是触摸点个数。最后一个函数是 ft5426_read_tpcoord，此函数就是读取 FT5426 各个触摸点坐标值的。

最后在 main.c 中输入如下内容：

示例代码 28.3.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : main.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 20 触摸屏实验
其他      : I.MX6U-ALPHAL 推荐使用正点原子-7 寸 LCD，此款 LCD 支持 5 点电容触摸，
            本节我们就来学习如何驱动 LCD 上的 5 点电容触摸屏。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/21 左忠凯创建
***************************************************************/
1  #include "bsp_clk.h"
```

```
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "bsp_lcd.h"
9  #include "bsp_lcdapi.h"
10 #include "bsp_rtc.h"
11 #include "bsp_ft5xx6.h"
12 #include "stdio.h"
13 
14 /*
15  * @description    : 使能 I.MX6U 的硬件 NEON 和 FPU
16  * @param          : 无
17  * @return         : 无
18  */
19  void imx6ul_hardfpu_enable(void)
20 {
21     uint32_t cpacr;
22     uint32_t fpexc;
23 
24     /* 使能 NEON 和 FPU */
25     cpacr = __get_CPACR();
26     cpacr = (cpacr & ~(CPACR_ASEDIS_Msk | CPACR_D32DIS_Msk))
27            |  (3UL << CPACR_cp10_Pos) | (3UL << CPACR_cp11_Pos);
28     __set_CPACR(cpacr);
29     fpexc = __get_FPEXC();
30     fpexc |= 0x40000000UL;
31     __set_FPEXC(fpexc);
32 }
33 
34 /*
35  * @description    : main 函数
36  * @param          : 无
37  * @return         : 无
38  */
39 int main(void)
40 {
41     unsigned char i = 0;
42     unsigned char state = OFF;
43 
44     imx6ul_hardfpu_enable();    /* 使能 I.MX6U 的硬件浮点    */
```

```
45      int_init();                 /* 初始化中断(一定要最先调用！) */
46      imx6u_clkinit();            /* 初始化系统时钟           */
47      delay_init();               /* 初始化延时               */
48      clk_enable();               /* 使能所有的时钟           */
49      led_init();                 /* 初始化 led               */
50      beep_init();                /* 初始化 beep              */
51      uart_init();                /* 初始化串口，波特率 115200 */
52      lcd_init();                 /* 初始化 LCD               */
53      ft5426_init();              /* 初始化触摸屏             */
54
55      tftlcd_dev.forecolor = LCD_RED;
56      lcd_show_string(50, 10, 400, 24, 24,
                     (char*)"ALPHA-IMX6U TOUCH SCREEN TEST");
57      lcd_show_string(50, 40, 200, 16, 16,
                    (char*)"TOUCH SCREEN TEST");
58      lcd_show_string(50, 60, 200, 16, 16, (char*)"ATOM@ALIENTEK");
59      lcd_show_string(50, 80, 200, 16, 16, (char*)"2019/3/27");
60      lcd_show_string(50, 110, 400, 16, 16,  (char*)"TP Num  :");
61      lcd_show_string(50, 130, 200, 16, 16,  (char*)"Point0 X:");
62      lcd_show_string(50, 150, 200, 16, 16,  (char*)"Point0 Y:");
63      lcd_show_string(50, 170, 200, 16, 16,  (char*)"Point1 X:");
64      lcd_show_string(50, 190, 200, 16, 16,  (char*)"Point1 Y:");
65      lcd_show_string(50, 210, 200, 16, 16,  (char*)"Point2 X:");
66      lcd_show_string(50, 230, 200, 16, 16,  (char*)"Point2 Y:");
67      lcd_show_string(50, 250, 200, 16, 16,  (char*)"Point3 X:");
68      lcd_show_string(50, 270, 200, 16, 16,  (char*)"Point3 Y:");
69      lcd_show_string(50, 290, 200, 16, 16,  (char*)"Point4 X:");
70      lcd_show_string(50, 310, 200, 16, 16,  (char*)"Point4 Y:");
71      tftlcd_dev.forecolor = LCD_BLUE;
72      while(1)
73      {
74          lcd_shownum(50 + 72, 110, ft5426_dev.point_num , 1, 16);
75          lcd_shownum(50 + 72, 130, ft5426_dev.x[0], 5, 16);
76          lcd_shownum(50 + 72, 150, ft5426_dev.y[0], 5, 16);
77          lcd_shownum(50 + 72, 170, ft5426_dev.x[1], 5, 16);
78          lcd_shownum(50 + 72, 190, ft5426_dev.y[1], 5, 16);
79          lcd_shownum(50 + 72, 210, ft5426_dev.x[2], 5, 16);
80          lcd_shownum(50 + 72, 230, ft5426_dev.y[2], 5, 16);
81          lcd_shownum(50 + 72, 250, ft5426_dev.x[3], 5, 16);
82          lcd_shownum(50 + 72, 270, ft5426_dev.y[3], 5, 16);
83          lcd_shownum(50 + 72, 290, ft5426_dev.x[4], 5, 16);
84          lcd_shownum(50 + 72, 310, ft5426_dev.y[4], 5, 16);
85
```

```
86          delayms(10);
87          i++;
88
89          if(i == 50)
90          {
91              i = 0;
92              state = !state;
93              led_switch(LED0,state);
94          }
95      }
96      return 0;
97 }
```

文件 main.c 只第 53 行调用函数 ft5426_init 初始化触摸屏，也就是 FT5426 这个触摸驱动 IC。最后在 main 函数的 while 循环中不断的显示获取到的触摸点数以及对应的触摸坐标值。因为本章实验我们采用中断方式读取 FT5426 的触摸数据，因此 main 函数中并没有读取 FT5426 的操作，只是显示触摸值。本章实验程序编写就到这里，接下来就是编译、下载和验证。

28.4 编译下载验证

28.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 touchscreen，然后在 INCDIRS 和 SRCDIRS 中加入“bsp/touchscreen”，修改后的 Makefile 如下：

示例代码 28.4.1.1 Makefile 文件代码

```
1  CROSS_COMPILE   ?= arm-linux-gnueabihf-
2  TARGET          ?= touchscreen
3
4  /* 省略掉其它代码...... */
5
6  INCDIRS         :=  imx6ul \
7                      stdio/include \
8                      bsp/clk \
9                      bsp/led \
10                     bsp/delay  \
11                     bsp/beep \
12                     bsp/gpio \
13                     bsp/key \
14                     bsp/exit \
15                     bsp/int \
16                     bsp/epittimer \
17                     bsp/keyfilter \
18                     bsp/uart \
19                     bsp/lcd \
20                     bsp/rtc \
```

```
21                      bsp/i2c \
22                      bsp/ap3216c \
23                      bsp/spi \
24                      bsp/icm20608 \
25                      bsp/touchscreen
26
27 SRCDIRS          :=  project \
28                      stdio/lib \
29                      bsp/clk \
30                      bsp/led \
31                      bsp/delay \
32                      bsp/beep \
33                      bsp/gpio \
34                      bsp/key \
35                      bsp/exit \
36                      bsp/int \
37                      bsp/epittimer \
38                      bsp/keyfilter \
39                      bsp/uart \
40                      bsp/lcd \
41                      bsp/rtc \
42                      bsp/i2c \
43                      bsp/ap3216c \
44                      bsp/spi \
45                      bsp/icm20608 \
46                      bsp/touchscreen
47
48 /* 省略掉其它代码...... */
49
50 clean:
51  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“touchscreen”，也就是目标名称为“touchscreen”。

第 25 行在变量 INCDIRS 中添加触摸屏的驱动头文件(.h)路径。

第 46 行在变量 SRCDIRS 中添加触摸屏的驱动文件(.c)路径。

链接脚本保持不变。

28.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 touchscreen.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                         //给予 imxdownload 可执行权限，一次即可
./imxdownload touchscreen.bin /dev/sdd        //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板。默认情况下 LCD 界面如图 28.4.2.1 所示：

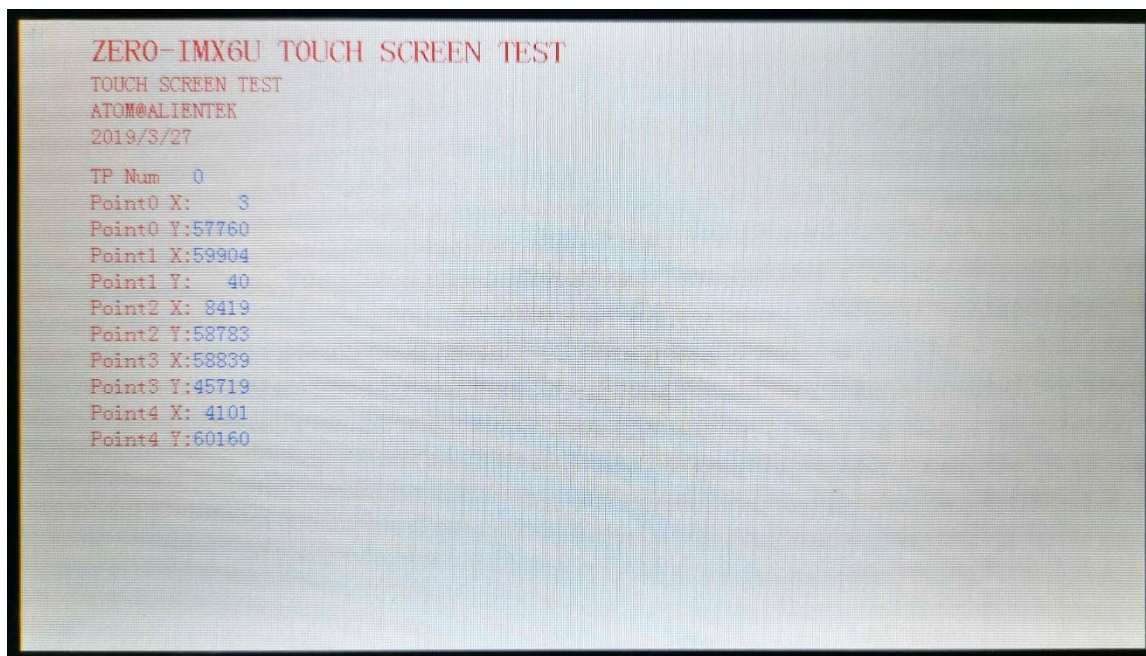


图 28.4.2.1 默认 LCD 显示

当我们用手指触摸屏幕的时候就会在 LCD 上显示出当前的触摸点和对应的触摸值，如图 28.4.2.2 所示：

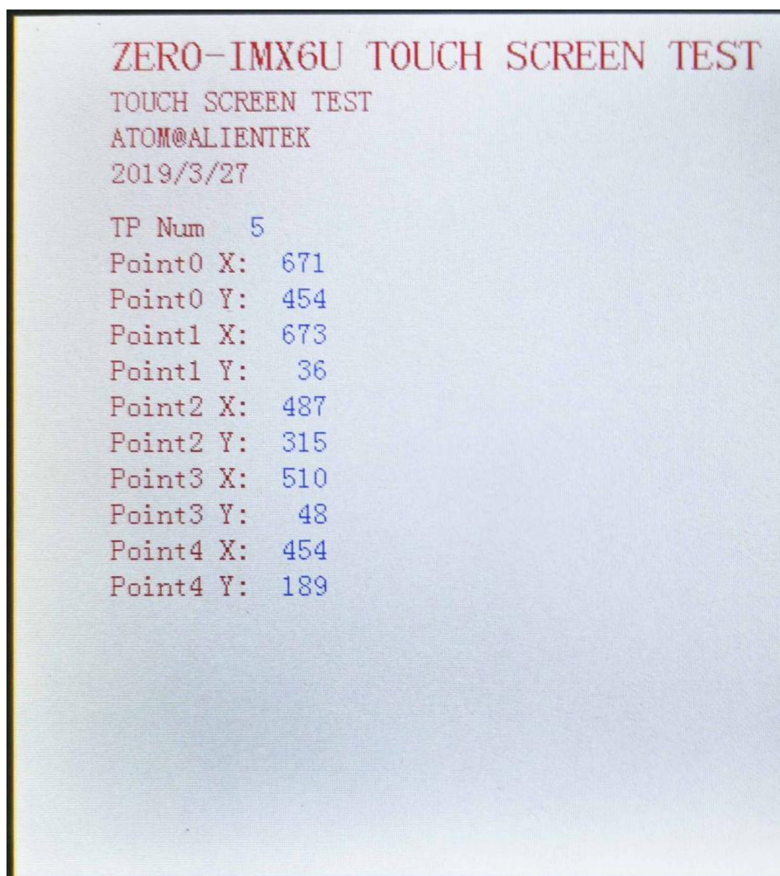


图 28.4.2.2

图 28.4.2.2 中有 5 个触摸点，每个触摸点的坐标全部显示到了 LCD 屏幕上。如果移动手指的话 LCD 上的触摸点坐标数据就会相应的变化。

第二十九章 LCD 背光调节实验

不管是使用显示器还是手机，其屏幕背光都是可以调节的，通过调节背光就可以控制屏幕的亮度。在户外阳光强烈的时候可以通过调高背光来看清屏幕，在光线比较暗的地方可以调低背光，防止伤眼睛并且省电。正点原子的三款 RGB LCD 也支持背光调节，本章我们就来学习如何调节 LCD 背光。

29.1 LCD 背光调节简介

正点原子的三个 RGB LCD 都有一个背光控制引脚，给这个背光控制引脚输入高电平就会点亮背光，输入低电平就会关闭背光。假如我们不断的打开和关闭背光，当速度足够快的时候就不会感觉到背光关闭这个过程了。这个正好可以使用 PWM 来完成，PWM 全称是 Pulse Width Modulation，也就是脉冲宽度调制，PWM 信号如图 29.1.1 所示：

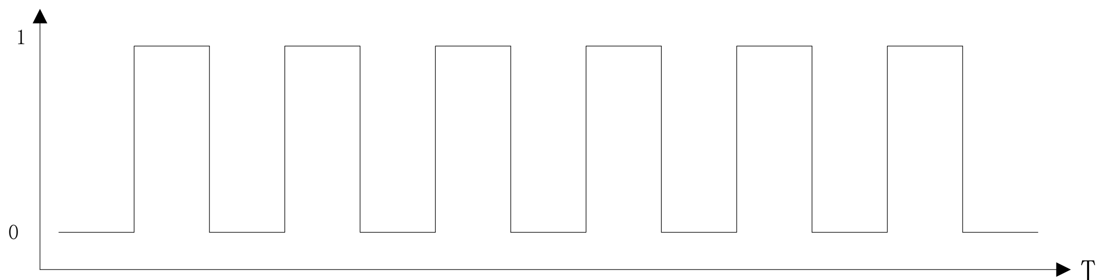


图 29.1.1 PWM 信号

PWM 信号有两个关键的术语：频率和占空比，频率就是开关速度，把一次开关算作一个周期，那么频率就是 1 秒内进行了多少次开关。占空比就是一个周期内高电平时间和低电平时间的比例，一个周期内高电平时间越长占空比就越大，反之占空比就越小。占空比用百分之表示，如果一个周期内全是低电平那么占空比就是 0%，如果一个周期内全是高电平那么占空比就是 100%。

我们给 LCD 的背光引脚输入一个 PWM 信号，这样就可以通过调整占空比的方式来调整 LCD 背光亮度了。提高占空比就会提高背光亮度，降低占空比就会降低背光亮度。重点就在于 PWM 信号的产生和占空比的控制，很幸运的是，I.MX6U 提供了 PWM 外设，因此我们可以配置 PWM 外设来产生 PWM 信号。

打开《I.MX6ULL 参考手册》的第 40 章“Chapter 40 Pulse Width Modulation(PWM)”，I.MX6U 一共有 8 路 PWM 信号，每个 PWM 包含一个 16 位的计数器和一个 4 x 16 的数据 FIFO，I.MX6U 的 PWM 外设结构如图 29.1.2 所示：

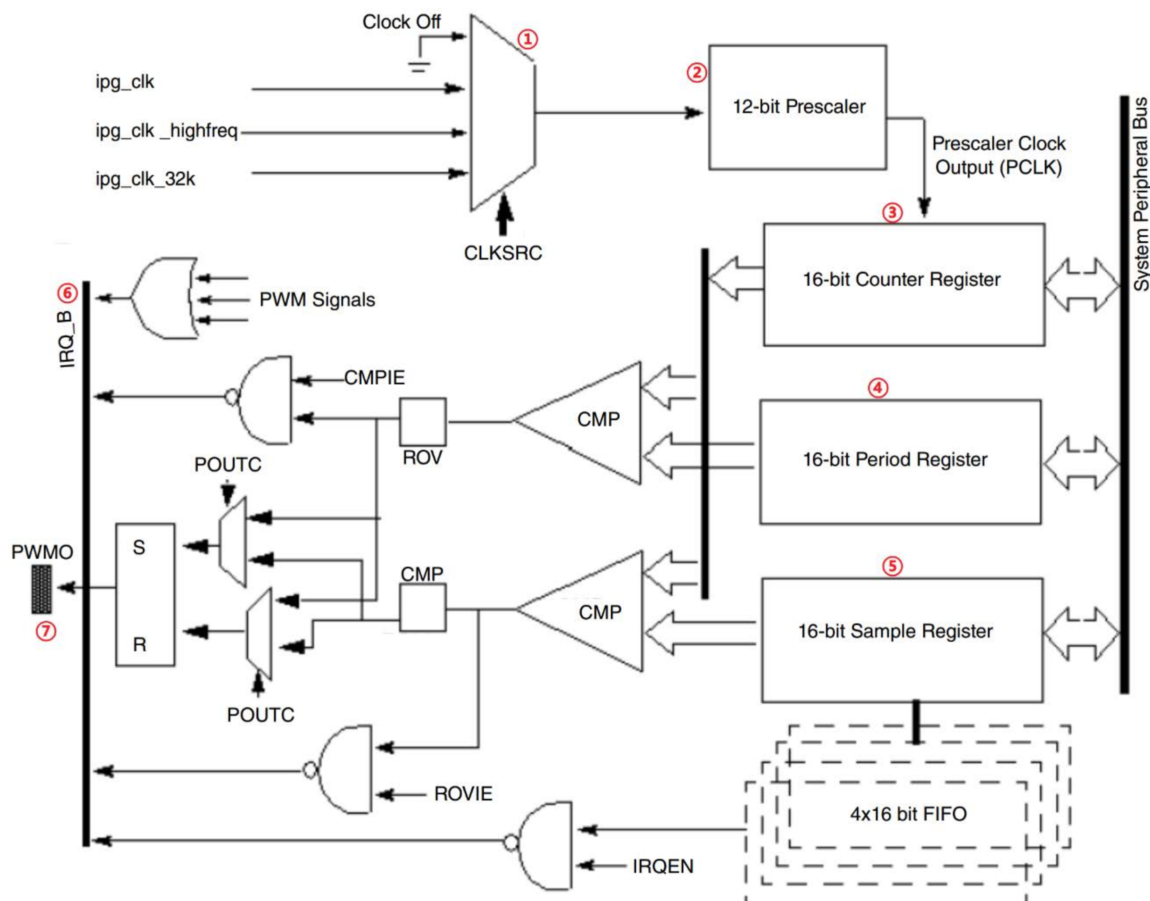


图 29.1.2 I.MX6U PWM 结构框图

图 29.1.2 中的各部分功能如下：

①、此部分是一个选择器，用于选择 PWM 信号的时钟源，一共有三种时钟源：ipg_clk、ipg_clk_highfreq 和 ipg_clk_32k。

②、这是一个 12 位的分频器，可以对①中选择的时钟源进行分频。

③、这是 PWM 的 16 位计数器寄存器，保存着 PWM 的计数值。

④、这是 PWM 的 16 位周期寄存器，此寄存器用来控制 PWM 的频率。

⑤、这是 PWM 的 16 位采样寄存器，此寄存器用来控制 PWM 的占空比。

⑥、此部分是 PWM 的中断信号，PWM 是提供中断功能的，如果使能了相应的中断的话就会产生中断。

⑦、此部分是 PWM 对应的输出 IO，产生的 PWM 信号就会从对应的 IO 中输出，I.MX6U-ALPHA 开发板的 LCD 背光控制引脚连接在 I.MX6U 的 GPIO1_IO8 上，GPIO1_IO8 可以复用为 PWM1_OUT。

可以通过配置相应的寄存器来设置 PWM 信号的频率和占空比，PWM 的 16 位计数器是个向上计数器，此计数器会从 0X0000 开始计数，直到计数值等于寄存器 PWMx_PWMPR(x=1~8) + 1，然后计数器就会重新从 0X0000 开始计数，如此往复。所以寄存器 PWMx_PWMPR 可以设置 PWM 的频率。

在一个周期内，PWM 从 0X0000 开始计数的时候，PWM 引脚先输出高电平(默认情况下，可以通过配置输出低电平)。采样 FIFO 中保存的采样值会在每个时钟和计数器值进行比较，当采样值和计数器相等的话 PWM 引脚就会改为输出低电平(默认情况下，同样可以通过配置输出高电平)。计数器会持续计数，直到和周期寄存器 PWMx_PWMPR(x=1~8) + 1 的值相等，这样

一个周期就完成了。所以，采样 FIFO 控制着占空比，而采样 FIFO 里面的值来源于采样寄存器 PWMx_PWMSAR，因此相当于 PWMx_PWMSAR 控制着占空比。至此，PWM 信号的频率和占空比设置我们就知道该如何去做了。

PWM 开启以后会按照默认值运行，并产生 PWM 波形，而这个默认的 PWM 一般并不是我们需要的波形。如果这个 PWM 波形控制着设备的话就会导致设备因为接收到错误的 PWM 信号而运行错误，严重情况下可能会损坏设备，甚至人身安全。因此，在开启 PWM 之前最好设置好 PWMx_PWMPR 和 PWMx_PWMSAR 这两个寄存器，也就是设置好 PWM 的频率和占空比。

当我们向 PWMx_PWMSAR 寄存器写入采样值的时候，如果 FIFO 没满的话其值会被存储到 FIFO 中。如果 FIFO 满的时候写入采样值就会导致寄存器 PWMx_PWMSR 的位 FWE(bit6)置 1，表示 FIFO 写错误，FIFO 里面的值也并不会改变。FIFO 可以在任何时候写入，但是只有在 PWM 使能的情况下读取。寄存器 PWMx_SR 的位 FIFOAV(bit2:0)记录着当前 FIFO 中有多少个数据。从采样寄存器 PWMx_PWMSAR 读取一次数据，FIFO 里面的数据就会减一，每产生一个周期的 PWM 信号，FIFO 里面的数据就会减一，相当于被用掉了。PWM 有个 FIFO 空中断，当 FIFO 为空的时候就会触发此中断，可以在此中断处理函数中向 FIFO 写入数据。

关于 I.MX6U 的 PWM 的原理知识就讲解到这里，接下来看一下 PWM 的几个重要的寄存器，本章我们使用的是 PWM1，首先看一下寄存器 PWM1_PWMCR 寄存器，此寄存器结构如图 29.1.2 所示：

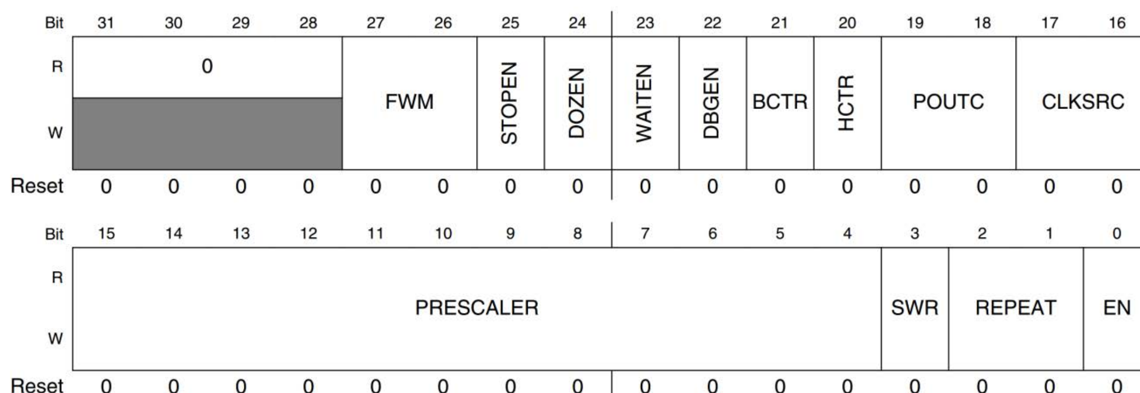


图 29.1.2 寄存器 PWM1_PWMCR 寄存器结构

寄存器 PWM1_PWMCR 用到的重要位如下：

FWM(bit27:26)：FIFO 水位线，用来设置 FIFO 空余位置为多少的时候表示 FIFO 为空。设置为 0 的时候表示 FIFO 空余位置大于等于 1 的时候 FIFO 为空；设置为 1 的时候表示 FIFO 空余位置大于等于 2 的时候 FIFO 为空；设置为 2 的时候表示 FIFO 空余位置大于等于 3 的时候 FIFO 为空；设置为 3 的时候表示 FIFO 口语位置大于等于 4 的时候 FIFO 为空。

STOPEN(bit25)：此位用来设置停止模式下 PWM 是否工作，为 0 的话表示在停止模式下 PWM 继续工作，为 1 的话表示停止模式下关闭 PWM。

DOZEN(bit24)：此位用来设置休眠模式下 PWM 是否工作，为 0 的话表示在休眠模式下 PWM 继续工作，为 1 的话表示休眠模式下关闭 PWM。

WAITEN(bit23)：此位用来设置等待模式下 PWM 是否工作，为 0 的话表示在等待模式下 PWM 继续工作，为 1 的话表示等待模式下关闭 PWM。

DEGEN(bit22)：此位用来设置调试模式下 PWM 是否工作，为 0 的话表示在调试模式下 PWM 继续工作，为 1 的话表示调试模式下关闭 PWM。

BCTR(bit21)：字节交换控制位，用来控制 16 位的数据进入 FIFO 的字节顺序。为 0 的时候不进行字节交换，为 1 的时候进行字节交换。

HCRT(bit20)：半字交换控制位，用来决定从 32 位 IP 总线接口传输来的哪个半字数据写入采样寄存器的低 16 位中。

POUTC(bit19:18)：PWM 输出控制控制位，用来设置 PWM 输出模式，为 0 的时候表示 PWM 先输出高电平，当计数器值和采样值相等的话就输出低电平。为 1 的时候相反，当为 2 或者 3 的时候 PWM 信号不输出。本章我们设置为 0，也就是一开始输出高电平，当计数器值和采样值相等的话就改为低电平，这样采样值越大高电平时间就越长，占空比就越大。

CLKSRC(bit17:16)：PWM 时钟源选择，为 0 的话关闭；为 1 的话选择 ipg_clk 为时钟源；为 2 的话选择 ipg_clk_highfreq 为时钟源；为 3 的话选择 ipg_clk_32k 为时钟源。本章我们设置为 1，也就是选择 ipg_clk 为 PWM 的时钟源，因此 PWM 时钟源频率为 66MHz。

PRESCALER(bit15:4)：分频值，可设置为 0~4095，对应着 1~4096 分频。

SWR(bit3)：软件复位，向此位写 1 就复位 PWM，此位是自清零的，当复位完成以后此位会自动清零。

REPEAT(bit2:1)：重复采样设置，此位用来设置 FIFO 中的每个数据能用几次。可设置 0~3，分别表示 FIFO 中的每个数据能用 1~4 次。本章我们设置为 0，即 FIFO 只的每个数据只能用一次。

EN(bit0)：PWM 使能位，为 1 的时候使能 PWM，为 0 的时候关闭 PWM。

接下来看一下寄存器 PWM1_PWMIR 寄存器，这个是 PWM 的中断控制寄存器，此寄存器结构如图 29.1.3 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0													CIE	RIE	FIE
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

图 29.1.3 寄存器 PWM1_PWMIR 结构

寄存器 PWM1_PWMIR 只有三个位，这三个位的含义如下：

CIE(bit2)：比较中断使能位，为 1 的时候使能比较中断，为 0 的时候关闭比较中断。

RIE(bit1)：翻转中断使能位，当计数器值等于采样值并回滚到 0X0000 的时候就会产生此中断，为 1 的时候使能翻转中断，为 0 的时候关闭翻转中断。

FIE(bit0)：FIFO 空中断，为 1 的时候使能，为 0 的时候关闭。

再来看一下状态寄存器 PWM1_PWMSR，此寄存器结构如图 29.1.4 所示：

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
R	0															
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	0									FWE	CMP	ROV	FE	FIFOAV		
W										w1c	w1c	w1c	w1c			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

图 29.1.4 寄存器 PWM1_PWMSR 结构

寄存器 PWM1_PWMSR 各个位的含义如下：

FWE(bit6)： FIFO 写错误事件，为 1 的时候表示发生了 FIFO 写错误。

CMP(bit5)： FIFO 比较事件发标志位，为 1 的时候表示发生 FIFO 比较事件。

ROV(bit4)： 翻转事件标志位，为 1 的话表示翻转事件发生。

FE(bit3)： FIFO 空标志位，为 1 的时候表示 FIFO 位空。

FIFOAV(bit2:1)： 此位记录 FIFO 中的有效数据个数，有效值为 0~4，分别表示 FIFO 中有 0~4 个有效数据。

接下来是寄存器 PWM1_PWMPR 寄存器，这个是 PWM 周期寄存器，可以通过此寄存器来设置 PWM 的频率，此寄存器结构如图 29.1.5 所示：

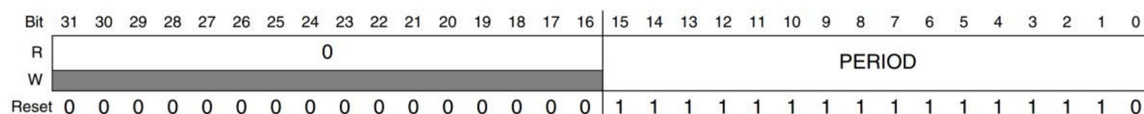


图 29.1.5 寄存器 PWM1_PWMPR 寄存器

从图 29.1.5 可以看出，寄存器 PWM1_PWMPR 只有低 16 位有效，当 PWM 计数器的值等于 PERIOD+1 的时候就会从 0X0000 重新开始计数，开启另一个周期。PWM 的频率计算公式如下：

$$PWMO(Hz) = PCLK(Hz) / (PERIOD + 2)$$

其中 PCLK 是最终进入 PWM 的时钟频率，假如 PCLK 的频率为 1MHz，现在我们要产生一个频率为 1KHz 的 PWM 信号，那么就可以设置 PERIOD = 1000000 / 1000 – 2 = 998。

最后来看一下寄存器 PWM1_PWMSAR，这是采样寄存器，用于设置占空比的，此寄存器结构如图 29.1.6 所示：

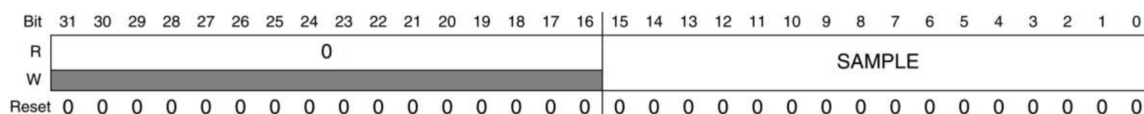


图 29.1.6 寄存器 PWM1_PWMSAR 结构

此寄存器也是只有低 16 位有效，为采样值。通过这个采样值即可调整占空比，当计数器的值小于 SAMPLE 的时候输出高电平(或低电平)。当计数器值大于等于 SAMPLE，小于寄存器 PWM1_PWMPR 的 PERIO 的时候输出低电平(或高电平)。同样在上面的例子中，假如我们要设置 PWM 信号的占空比为 50%，那么就可以将 SAMPLE 设置为(PERIOD + 2) / 2 = 1000 / 2=500。

关于 PWM 有关的寄存器就介绍到这里，关于这些寄存器详细的描述，请参考《I.MX6ULL 参考手册》第 2480 页的 40.7 小节。本章我们使用 I.MX6U 的 PWM1，PWM1 的输出引脚为 GPIO1_IO8，配置步骤如下：

1、配置引脚 GPIO1_IO8

配置 GPIO1_IO08 的复用功能，将其复用为 PWM1_OUT 信号线。

2、初始化 PWM1

初始化 PWM1，配置所需的 PWM 信号的频率和默认占空比。

3、设置中断

因为 FIFO 中的采样值每个周期都会少一个，所以需要不断的向 FIFO 中写入采样值，防止其为空。我们可以使能 FIFO 空中断，这样当 FIFO 为空的时候就会触发相应的中断，然后在中断处理函数中向 FIFO 写入采样值。

4、使能 PWM1

配置好 PWM1 以后就可以开启了。

29.2 硬件原理分析

本试验用到的资源如下：

、指示灯 LED0。

、RGB LCD 接口。

③、按键 KEY0

本实验用到的硬件原理图参考第二十四章，本章实验我们一开始设置 RGB LCD 的背光亮度 PWM 信号频率为 1KHz,占空比为 10%，这样屏幕亮度就很低。然后通过按键 KEY0 逐步的提升 PWM 信号的占空比，按照 10%步进。当达到 100%以后再次按下 KEY0，PWM 信号占空比回到 10%重新开始。LED0 不断的闪烁，提示系统正在运行。

29.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 1、裸机例程-> 20_pwm_lcdbacklight。

本章实验在上一章例程的基础上完成，更改工程名字为“backlight”，然后在 bsp 文件夹下创建名为“backlight”的文件夹，然后在 bsp/backlight 中新建 bsp_backlight.c 和 bsp_backlight.h 这两个文件。在 bsp_backlight.h 中输入如下内容：

示例代码 29.3.1 bsp_backlight.h 文件代码

```
1  #ifndef _BACKLIGHT_H
2  #define _BACKLIGHT_H
3  /***************************************************************
4  Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
5  文件名    : bsp_backlight.c
6  作者      : 左忠凯
7  版本      : V1.0
8  描述      : LCD 背光 PWM 驱动头文件。
9  其他      : 无
10 论坛      : www.openedv.com
11 日志      : 初版 V1.0 2019/1/22 左忠凯创建
12 ***************************************************************/
13 #include "imx6ul.h"
14 
15 /* 背光 PWM 结构体 */
16 struct backlight_dev_struc
17 {
18     unsigned char pwm_duty;    /* 占空比   */
19 };
20 
21 /* 函数声明 */
22 void backlight_init(void);
23 void pwm1_enable(void);
24 void pwm1_setsample_value(unsigned int value);
25 void pwm1_setperiod_value(unsigned int value);
26 void pwm1_setduty(unsigned char duty);
```

```
27 void pwm1_irqhandler(void);
28
29 #endif
```

文件 bsp_backlight.h 文件内容很简单，在第 16 行定义了一个背光 PWM 结构体，剩下的就是函数声明。在文件 bsp_backlight.c 中输入如下内容：

示例代码 29.3.2 bsp_backlight.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名	: bsp_backlight.c
作者	: 左忠凯
版本	: V1.0
描述	: LCD 背光 PWM 驱动文件。
其他	: 无
论坛	: www.openedv.com
日志	: 初版 V1.0 2019/1/22 左忠凯创建
***************************************************************/
1  #include "bsp_backlight.h"
2  #include "bsp_int.h"
3  #include "stdio.h"
4
5  struct backlight_dev_struc backlight_dev; /* 背光设备 */
6
7  /*
8   * @description	: pwm1 中断处理函数
9   * @param		: 无
10  * @return		: 无
11  */
12 void pwm1_irqhandler(void)
13 {
14     if(PWM1->PWMSR & (1 << 3))      /* FIFO 为空中断       */
15     {
16         /* 将占空比信息写入到 FIFO 中,其实就是设置占空比 */
17         pwm1_setduty(backlight_dev.pwm_duty);
18         PWM1->PWMSR |= (1 << 3);    /* 写 1 清除中断标志位   */
19     }
20 }
21
22 /*
23  * @description	: 初始化背光 PWM
24  * @param		: 无
25  * @return		: 无
26  */
27 void backlight_init(void)
```

```
28 {
29     unsigned char i = 0;
30 
31     /* 1、背光 PWM IO 初始化,复用为 PWM1_OUT */
32     IOMUXC_SetPinMux(IOMUXC_GPIO1_IO08_PWM1_OUT, 0);
33     IOMUXC_SetPinConfig(IOMUXC_GPIO1_IO08_PWM1_OUT, 0XB090);
34 
35     /* 2、初始化 PWM1
36      * 初始化寄存器 PWMCR
37      * bit[27:26] : 01  当 FIFO 中空余位置大于等于 2 的时候 FIFO 空标志值位
38      * bit[25]    : 0  停止模式下 PWM 不工作
39      * bit[24]    : 0   休眠模式下 PWM 不工作
40      * bit[23]    : 0   等待模式下 PWM 不工作
41      * bit[22]    : 0   调试模式下 PWM 不工作
42      * bit[21]    : 0   关闭字节交换
43      * bit[20]    : 0   关闭半字数据交换
44      * bit[19:18] : 00  PWM 输出引脚在计数器重新计数的时候输出高电平
45      *                  在计数器计数值达到比较值以后输出低电平
46      * bit[17:16] : 01  PWM 时钟源选择 IPG CLK = 66MHz
47      * bit[15:4]  : 65  分频系数为 65+1=66，PWM 时钟源 = 66MHZ/66=1MHz
48      * bit[3]     : 0   PWM 不复位
49      * bit[2:1]   : 00  FIFO 中的 sample 数据每个只能使用一次。
50      * bit[0]     : 0   先关闭 PWM，后面再使能
51      */
52     PWM1->PWMCR = 0;    /* 寄存器先清零 */
53     PWM1->PWMCR |= (1 << 26) | (1 << 16) | (65 << 4);
54 
55     /* 设置 PWM 周期为 1000,那么 PWM 频率就是 1M/1000 = 1KHz。 */
56     pwm1_setperiod_value(1000);
57 
58     /* 设置占空比，默认 50%占空比  ,写四次是因为有 4 个 FIFO */
59     backlight_dev.pwm_duty = 50;
60     for(i = 0; i < 4; i++)
61     {
62         pwm1_setduty(backlight_dev.pwm_duty);
63     }
64 
65     /* 使能 FIFO 空中断，设置寄存器 PWMIR 寄存器的 bit0 为 1 */
66     PWM1->PWMIR |= 1 << 0;
67     system_register_irqhandler(PWM1_IRQn, /* 注册中断服务函数     */
                         (system_irq_handler_t)pwm1_irqhandler, NULL);
68     GIC_EnableIRQ(PWM1_IRQn);                /* 使能 GIC 中对应的中断 */
69     PWM1->PWMSR = 0;                         /* PWM 中断状态寄存器清零 */
```

```
70      pwm1_enable();                              /* 使能 PWM1          */
71  }
72
73  /*
74   * @description   : 使能 PWM
75   * @param         : 无
76   * @return        : 无
77   */
78  void pwm1_enable(void)
79  {
80      PWM1->PWMCR |= 1 << 0;
81  }
82
83  /*
84   * @description   : 设置 Sample 寄存器，Sample 数据会写入到 FIFO 中，所谓的
85   *                  Sample 寄存器，就相当于比较寄存器，假如 PWMCR 中的 POUTC
86   *                  设置为 00 的时候。当 PWM 计数器中的计数值小于 Sample 的时候
87   *                  就会输出高电平，当 PWM 计数器值大于 Sample 的时候输出底电
88   *                  平，因此可以通过设置 Sample 寄存器来设置占空比。
89   * @param - value: 寄存器值，范围 0~0XFFFF
90   * @return        : 无
91   */
92  void pwm1_setsample_value(unsigned int value)
93  {
94      PWM1->PWMSAR = (value & 0XFFFF);
95  }
96
97  /*
98   * @description   : 设置 PWM 周期，就是设置寄存器 PWMPR，PWM 周期公式如下
99   *             PWM_FRE = PWM_CLK / (PERIOD + 2)， 比如当前 PWM_CLK=1MHz
100  *             要产生 1KHz 的 PWM，那么 PERIOD = 1000000/1K - 2 = 998
101  * @param - value   : 周期值，范围 0~0XFFFF
102  * @return          : 无
103  */
104 void pwm1_setperiod_value(unsigned int value)
105 {
106     unsigned int regvalue = 0;
107
108     if(value < 2)
109         regvalue = 2;
110     else
111         regvalue = value - 2;
112     PWM1->PWMPR = (regvalue & 0XFFFF);
```

```
113 }
114
115 /*
116  * @description       : 设置 PWM 占空比
117  * @param -  value    : 占空比 0~100，对应 0%~100%
118  * @return            : 无
119  */
120 void pwm1_setduty(unsigned char duty)
121 {
122     unsigned short preiod;
123     unsigned short sample;
124
125     backlight_dev.pwm_duty = duty;
126     preiod = PWM1->PWMPR + 2;
127     sample = preiod * backlight_dev.pwm_duty / 100;
128     pwm1_setsample_value(sample);
129 }
```

文件 bsp_blacklight.c 一共有 6 个函数，首先是函数 pwm1_irqhandler，这个是 PWM1 的中断处理函数。需要在此函数中处理 FIFO 空中断，当 FIFO 空中断发生以后需要向采样寄存器 PWM1_PWMSAR 写入采样数据，也就是占空比值，最后要清除相应的中断标志位。第 2 个函数是 backlight_init，这个是背光初始化函数，在此函数里面会初始化背光引脚 GPIO1_IO08，将其复用为 PWM1_OUT。然后此函数初始化 PWM1，设置要产生的 PWM 信号频率和默认占空比，接下来使能 FIFO 空中断，注册相应的中断处理函数，最后使能 PWM1。第 3 个函数是 pwm1_enable，用于使能 PWM1。第 4 个函数是 pwm1_setsample_value，用于设置采样值，也就是寄存器 PWM1_PWMSAR 的值。第 5 个函数是 pwm1_setperiod_value，用于设置 PWM 信号的频率。第 6 个函数是 pwm1_setduty，用于设置 PWM 的占空比，这个函数只有一个参数 duty，也就是占空比值，单位为%，函数内部会根据百分值计算出寄存器 PWM1_PWMSAR 应该设置的值。

最后在 main.c 文件中输入如下所示内容：

示例代码 29.3.3 main.c 文件代码

```
/***************************************************************
Copyright © zuozhongkai Co., Ltd. 1998-2019. All rights reserved.
文件名    : mian.c
作者      : 左忠凯
版本      : V1.0
描述      : I.MX6U 开发板裸机实验 21 背光 PWM 实验
其他      : 我们使用手机的时候背光都是可以调节的，同样的 I.MX6U-ALPHA
           开发板的 LCD 背光也是可以调节，LCD 背光就相当于一个 LED 灯。
           LED 灯的亮灭可以通过 PWM 来控制，本实验我们就来学习一下如何
           通过 PWM 来控制 LCD 的背光。
论坛      : www.openedv.com
日志      : 初版 V1.0 2019/1/21 左忠凯创建
***************************************************************/
```

```
1  #include "bsp_clk.h"
2  #include "bsp_delay.h"
3  #include "bsp_led.h"
4  #include "bsp_beep.h"
5  #include "bsp_key.h"
6  #include "bsp_int.h"
7  #include "bsp_uart.h"
8  #include "bsp_lcd.h"
9  #include "bsp_lcdapi.h"
10 #include "bsp_rtc.h"
11 #include "bsp_backlight.h"
12 #include "stdio.h"
13 
14 /*
15  * @description    : main 函数
16  * @param          : 无
17  * @return         : 无
18  */
19 int main(void)
20 {
21     unsigned char keyvalue = 0;
22     unsigned char i = 0;
23     unsigned char state = OFF;
24     unsigned char duty = 0;
25 
26     int_init();                /* 初始化中断(一定要最先调用！) */
27     imx6u_clkinit();           /* 初始化系统时钟              */
28     delay_init();              /* 初始化延时                  */
29     clk_enable();              /* 使能所有的时钟              */
30     led_init();                /* 初始化 led                  */
31     beep_init();               /* 初始化 beep                 */
32     uart_init();               /* 初始化串口，波特率 115200   */
33     lcd_init();                /* 初始化 LCD                  */
34     backlight_init();          /* 初始化背光 PWM              */
35 
36     tftlcd_dev.forecolor = LCD_RED;
37     lcd_show_string(50, 10, 400, 24, 24,
               (char*)"ALPHA-IMX6U BACKLIGHT PWM TEST");
38     lcd_show_string(50, 40, 400, 24, 24, (char*)"PWM Duty:   %");
39     tftlcd_dev.forecolor = LCD_BLUE;
40 
41     /* 设置默认占空比 10% */
42     duty = 10;
```

```
43      lcd_shownum(158, 40, duty, 3, 24);
44      pwm1_setduty(duty);
45
46      while(1)
47      {
48          keyvalue = key_getvalue();
49          if(keyvalue == KEY0_VALUE)
50          {
51              duty += 10;          /* 占空比加 10% */
52              if(duty > 100)       /* 如果占空比超过 100%，重新从 10%开始 */
53                  duty = 10;
54              lcd_shownum(158, 40, duty, 3, 24);
55              pwm1_setduty(duty);  /* 设置占空比   */
56          }
57
58          delayms(10);
59          i++;
60          if(i == 50)
61          {
62              i = 0;
63              state = !state;
64              led_switch(LED0,state);
65          }
66      }
67      return 0;
68 }
```

第 34 行调用函数 backlight_init 初始化屏幕背光 PWM。第 44 行设置背光 PWM 默认占空比为 10%。在 main 函数中读取按键值，如果 KEY0 按下的话就将 PWM 信号的占空比增加 10%，当占空比超过 100%的时候就重回到 10%，重新开始。总的来说，main.c 的内容还是很简单的。

29.4 编译下载验证

29.4.1 编写 Makefile 和链接脚本

修改 Makefile 中的 TARGET 为 backlight，然后在在 INCDIRS 和 SRCDIRS 中加入“bsp/rtc”，修改后的 Makefile 如下：

示例代码 29.4.1 Makefile 代码

```
1  CROSS_COMPILE  ?= arm-linux-gnueabihf-
2  TARGET         ?= backlight
3
4  /* 省略掉其它代码...... */
5
6  INCDIRS         := imx6ul \
```

```
7                       stdio/include \
8                       bsp/clk \
9                       bsp/led \
10                      bsp/delay  \
11                      bsp/beep \
12                      bsp/gpio \
13                      bsp/key \
14                      bsp/exit \
15                      bsp/int \
16                      bsp/epittimer \
17                      bsp/keyfilter \
18                      bsp/uart \
19                      bsp/lcd \
20                      bsp/rtc \
21                      bsp/i2c \
22                      bsp/ap3216c \
23                      bsp/spi \
24                      bsp/icm20608 \
25                      bsp/touchscreen \
26                      bsp/backlight
27
28 SRCDIRS          :=  project \
29                      stdio/lib \
30                      bsp/clk \
31                      bsp/led \
32                      bsp/delay \
33                      bsp/beep \
34                      bsp/gpio \
35                      bsp/key \
36                      bsp/exit \
37                      bsp/int \
38                      bsp/epittimer \
39                      bsp/keyfilter \
40                      bsp/uart \
41                      bsp/lcd \
42                      bsp/rtc \
43                      bsp/i2c \
44                      bsp/ap3216c \
45                      bsp/spi \
46                      bsp/icm20608 \
47                      bsp/touchscreen \
48                      bsp/backlight
49
```

```
50 /* 省略掉其它代码...... */
51
52 clean:
53  rm -rf $(TARGET).elf $(TARGET).dis $(TARGET).bin $(COBJS) $(SOBJS)
```

第 2 行修改变量 TARGET 为“backlight”，也就是目标名称为“backlight”。

第 26 行在变量 INCDIRS 中添加背光 PWM 驱动头文件(.h)路径。

第 48 行在变量 SRCDIRS 中添加背光 PWM 驱动驱动文件(.c)路径。

链接脚本保持不变。

29.4.2 编译下载

使用 Make 命令编译代码，编译成功以后使用软件 imxdownload 将编译完成的 backlight.bin 文件下载到 SD 卡中，命令如下：

```
chmod 777 imxdownload                   //给予 imxdownload 可执行权限，一次即可
./imxdownload backlight.bin /dev/sdd    //烧写到 SD 卡中
```

烧写成功以后将 SD 卡插到开发板的 SD 卡槽中，然后复位开发板，默认背光 PWM 是 10%，PWM 信号波形如图 29.4.2.1 所示：

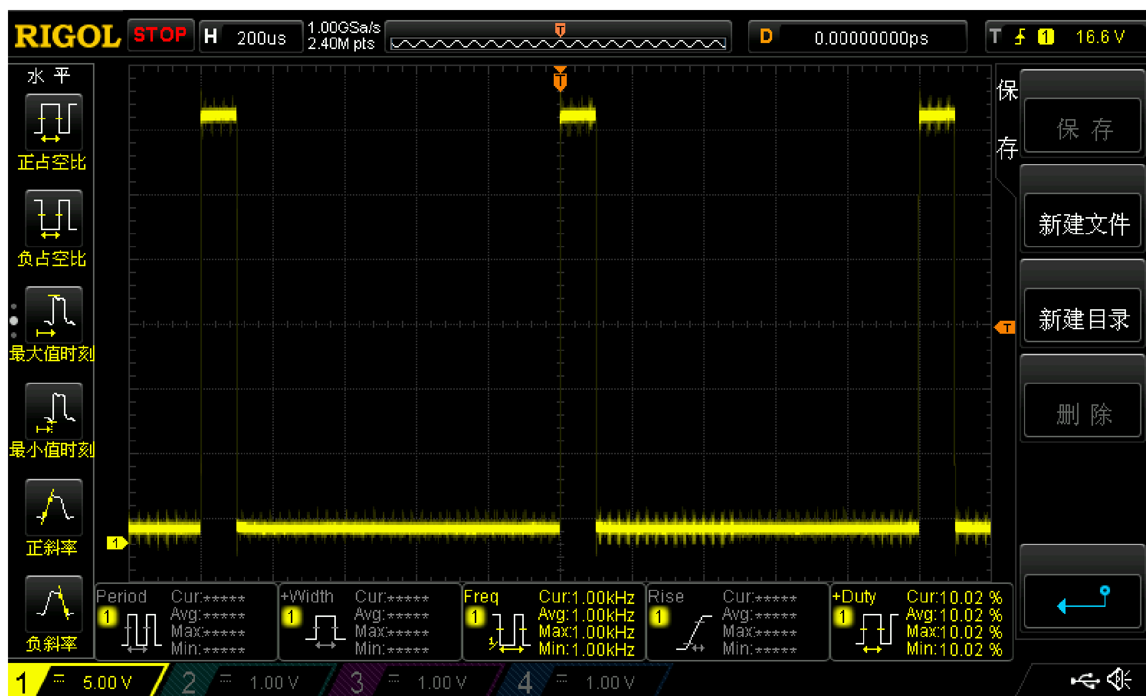


图 29.4.2.1 10%占空比 PWM 信号

从图 29.4.2.1 可以看出，此时背光 PWM 信号的频率为 1.00KHz，占空比是 10.02%，和我们代码中配置的一致，此时 LCD 的屏幕显示如图 29.4.2.2 所示：

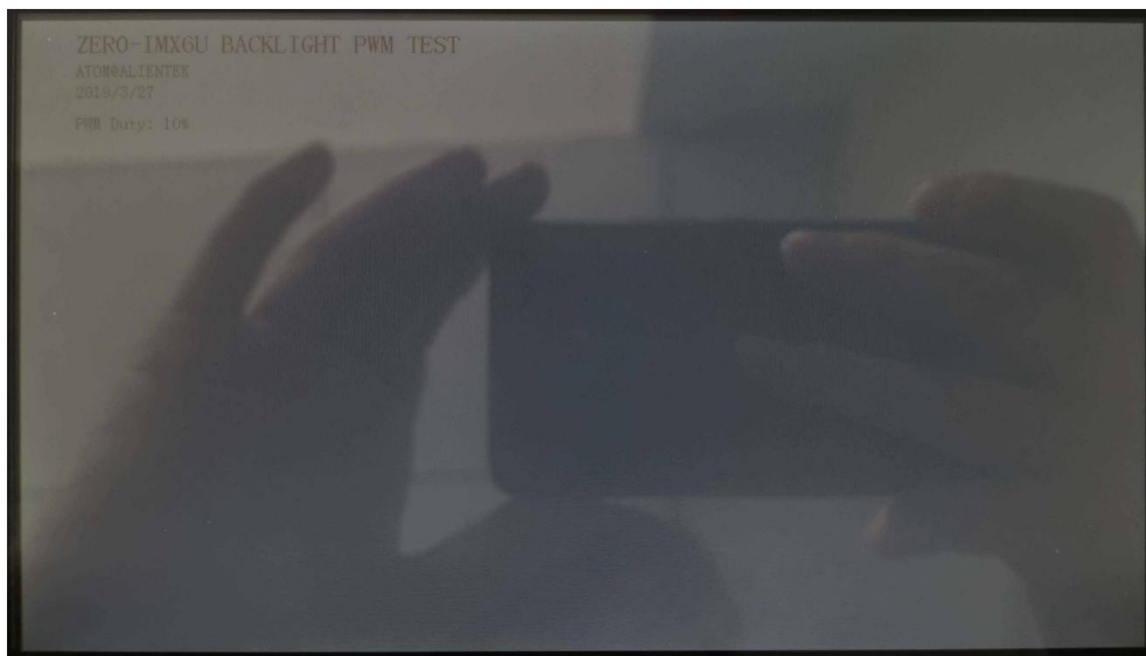


图 29.4.2.2 10%占空比屏幕亮度

图 29.4.2.2 就是 PWM 占空比为 10%的 LCD 屏幕显示，可以看出屏幕亮度很低，甚至可以看到屏幕上拍照人的倒影。因为拍照的原因，实际亮度跟实际情况可能会有少许差别。

我们将 PWM 的占空比调节到 90%，此时的 LCD 屏幕亮度肯定会很亮，如图 29.4.2.3 所示：

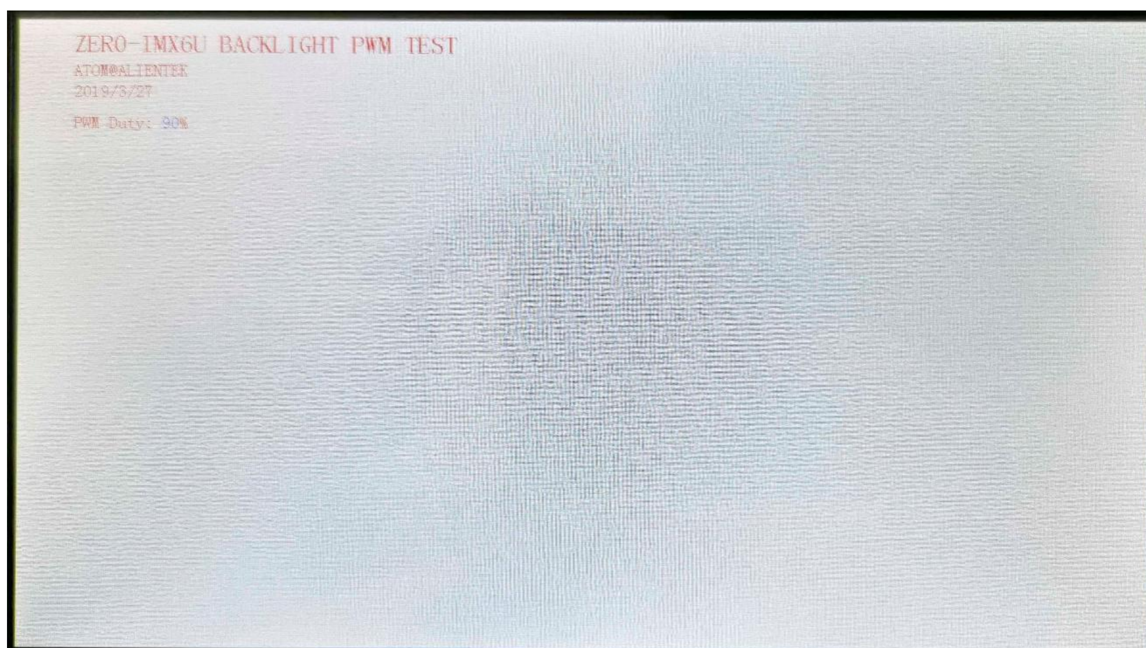


图 29.4.2.3 90%占空比屏幕亮度

图 29.4.2.3 的屏幕亮度相比图 29.4.2.2 就要高很多，这个就是 LCD 背光调节的原理，采用 PWM 波形来完成，通过调整占空比即可完成亮度调节。

至此，I.MX6U-ALPHA 开发板的所有裸机例程已经全部完成了，通过这几十个裸机例程，我们对 I.MX6UL/ULL 这款芯片的外设有了一个基本的了解，为我们以后学习 Uboot 和 Linux 驱动打下了坚实的基础。

第三篇 系统移植篇

在上一篇中我们学习了如何进行 I.MX6U 的裸机开发，通过 21 个裸机例程我们掌握了 I.MX6U 的常用外设。通过裸机的学习我们掌握了外设的底层原理，这样在以后进行 Linux 驱动开发的时候就只需要将精力放到 Linux 驱动框架上，在进行 Linux 驱动开发之前肯定需要先将 Linux 系统移植到开发板上去。如果学习过 UCOS/FreeRTOS 应该知道，UCOS/FreeRTOS 移植就是在官方的 SDK 包里面找一个和自己所使用的芯片一样的工程编译一下，然后下载到开发板就可以了。那么 Linux 的移植是不是也是这样的，下载 Linux 源码，然后找个和我们所使用的芯片一样的工程编译一下就可以了？很明显不是的！Linux 的移植要复杂的多，在移植 Linux 之前我们需要先移植一个 bootloader 代码，这个 bootloader 代码用于启动 Linux 内核，bootloader 有很多，常用的就是 U-Boot。移植好 U-Boot 以后再移植 Linux 内核，移植完 Linux 内核以后 Linux 还不能正常启动，还需要再移植一个根文件系统(rootfs)，根文件系统里面包含了一些最常用的命令和文件。所以 U-Boot、Linux kernel 和 rootfs 这三者一起构成了一个完整的 Linux 系统，一个可以正常使用、功能完善的 Linux 系统。在本篇我们就来讲解 U-Boot、Linux Kernel 和 rootfs 的移植，与其说是“移植”，倒不如说是“适配”，因为大部分的移植工作都由 NXP 完成了，我们这里所谓的“移植”主要是使其能够在 I.MX6U-ALPHA 开发板上跑起来。

第三十章 U-Boot 使用实验

在移植 U-Boot 之前，我们肯定要先使用一下 U-Boot，得先体验一下 U-Boot 是个什么东西。I.MX6U-ALPHA 开发板光盘资料里面已经提供了一个正点原子团队已经移植好的 U-Boot，本章我们就直接编译这个移植好的 U-Boot，然后烧写到 SD 卡里面启动，启动 U-Boot 以后就可以学习使用 U-Boot 的命令。

30.1 U-Boot 简介

Linux 系统要启动就必须需要一个 bootloader 程序，也就说芯片上电以后先运行一段 bootloader 程序。这段 bootloader 程序会先初始化 DDR 等外设，然后将 Linux 内核从 flash(NAND，NOR FLASH，SD，MMC 等)拷贝到 DDR 中，最后启动 Linux 内核。当然了，bootloader 的实际工作要复杂的多，但是它最主要的工作就是启动 Linux 内核，bootloader 和 Linux 内核的关系就跟 PC 上的 BIOS 和 Windows 的关系一样，bootloader 就相当于 BIOS。所以我们要先搞定 bootloader，很庆幸，有很多现成的 bootloader 软件可以使用，比如 U-Boot、vivi、RedBoot 等等，其中以 U-Boot 使用最为广泛，为了方便书写，本书会将 U-Boot 写为 uboot。

uboot 的全称是 Universal Boot Loader，uboot 是一个遵循 GPL 协议的开源软件，uboot 是一个裸机代码，可以看作是一个裸机综合例程。现在的 uboot 已经支持液晶屏、网络、USB 等高级功能。uboot 官网为 http://www.denx.de/wiki/U-Boot/，如图 30.1.1 所示：

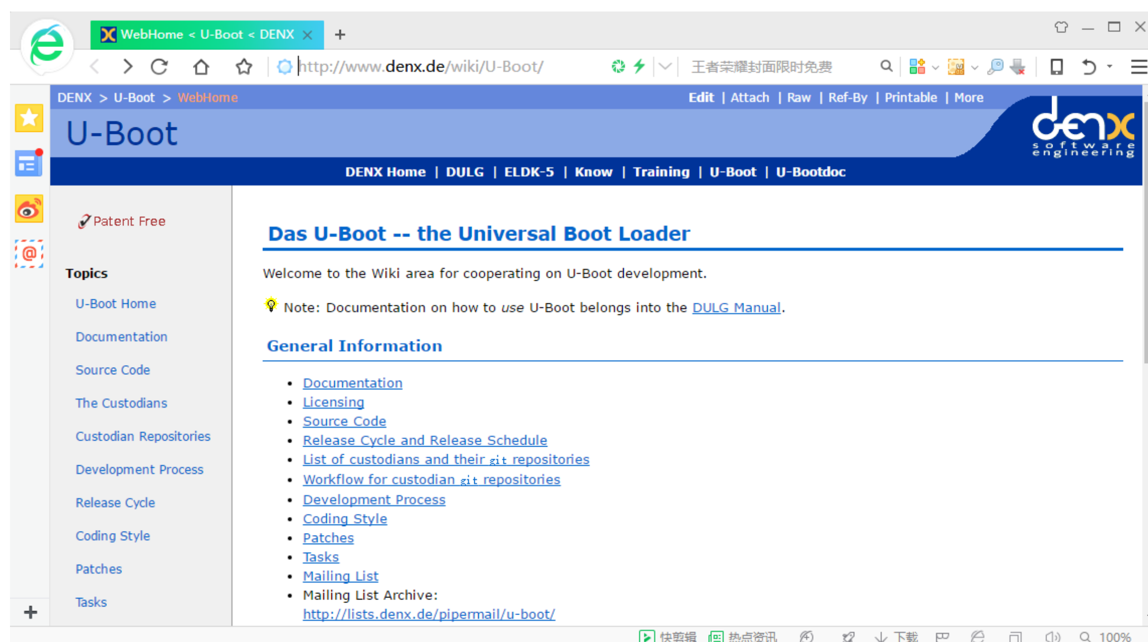


图 30.1.1 uboot 官网

我们可以在 uboot 官网下载 uboot 源码，点击图 30.1.1 中左侧 Topics 中的“Source Code”，打开如图 30.1.2 所示界面：



图 30.1.2 uboot 源码界面

点击图 30.1.2 中的“FTP Server”，进入其 FTP 服务器即可看到 uboot 源码，如图 30.1.3 所示：

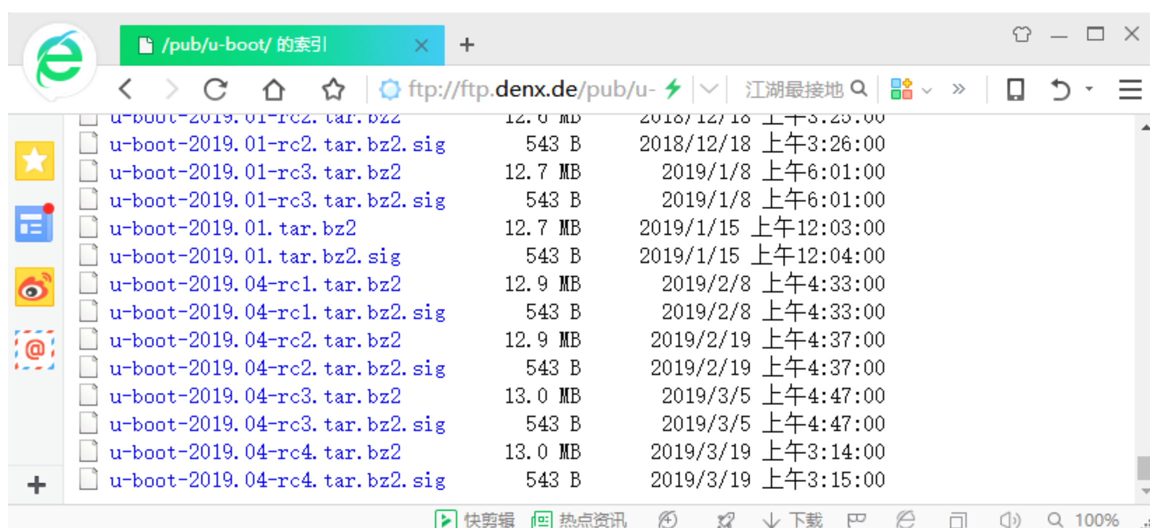


图 30.1.3 uboot 源码

图 30.1.3 中就是 uboot 原汁原味的源码文件，目前最新的版本是 2019.04。但是我们一般不会直接用 uboot 官方的 U-Boot 源码的。uboot 官方的 uboot 源码是给半导体厂商准备的，半导体厂商会下载 uboot 官方的 uboot 源码，然后将自家相应的芯片移植进去。也就是说半导体厂商会自己维护一个版本的 uboot，这个版本的 uboot 相当于是他们定制的。既然是定制的，那么肯定对自家的芯片支持会很全，虽然 uboot 官网的源码中一般也会支持他们的芯片，但是绝对是没有半导体厂商自己维护的 uboot 全面。

NXP 就维护的 2016.03 这个版本的 uboot，下载地址为：http://git.freescale.com/git/cgit.cgi/imx/uboot-imx.git/tag/?h=imx_v2016.03_4.1.15_2.0.0_ga&id=rel_imx_4.1.15_2.1.0_ga，下载界面如图 30.1.4 所示：



图 30.1.4 NXP 官方 uboot 下载界面

图 30.1.4 中的 uboot-imx_rel_imx4.1.15_2.1.0_ga.xx(xx 为 zip、tar.gz 或 tar.bz2)就是 NXP 官方维护的 uboott，后面我们学习 uboot 移植的时候就是使用的图 30.1.4 中的 uboot，下载 uboot-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2。我们已经放到了开发板光盘中，路径为：开发板光盘->1、程序源码->4、NXP 官方原版 Uboot 和 Linux->uboot-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2。图 30.1.4 中的 uboot 基本支持了 NXP 当前所有可以跑 Linux 的芯片，而且支持各种启动方式，比如 EMMC、NAND、NOR FLASH 等等，这些都是 uboot 官方所不支持的。但是图 30.1.4 中的 uboot 是针对 NXP 自家评估板的，如果是我们自己做的板子就需要修改 NXP 官方的 uboot，使其支持我们自己做的板子，正点原子的 I.MX6U 开发板就是自己做的板子，虽然大部分都参考了 NXP 官方的 I.MX6ULL EVK 开发板，但是还是有很多不同的地方，所以需要修改 NXP 官方的 uboot，使其适配正点原子的 I.MX6U 开发板。所以当我们拿到开发板以后，是有三种 uboot 的，这三种 uboot 的区别如表 30.1.1 所示：

种类	描述
uboot 官方的 uboot 代码	由 uboot 官方维护开发的 uboot 版本，版本更新快，基本包含所有常用的芯片。
半导体厂商的 uboot 代码	半导体厂商维护的一个 uboot，专门针对自家的芯片，在对自家芯片支持上要比 uboot 官方的好。
开发板厂商的 uboot 代码	开发板厂商在半导体厂商提供的 uboot 基础上加入了对自家开发板的支持。

表 30.1.1 三种 uboot 的区别

那么这三种 uboot 该如何选择呢？首先 uboot 官方的基本是不会用的，因为支持太弱了。最常用的就是半导体厂商或者开发板厂商的 uboot，如果你用的半导体厂商的评估板，那么就使用半导体厂商的 uboot，如果你是购买的第三方开发板，比如正点原子的 I.MX6ULL 开发板，那么就使用正点原子提供的 uboot 源码（也是在半导体厂商的 uboot 上修改的）。当然了，你也可以在购买了第三方开发板以后使用半导体厂商提供的 uboot，只不过有些外设驱动可能不支持，需要自己移植，这个就是我们常说的 uboot 移植。

本节是 uboot 的使用，所以就直接使用正点原子已经移植好的 uboot，这个已经放到了开发板光盘中了，路径为：开发板光盘->1、程序源码->3、正点原子修改后的 Uboot 和 Linux->uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek.tar.bz2。

30.2 U-Boot 初次编译

在 Ubuntu 中创建存放 uboot 的目录，比如我的是/home/$USER/linux/uboot，然后在此目录下新建一个名为“alientek_uboot”的文件夹用于存放正点原子提供的 uboot 源码。alientek_uboot 文件夹创建成功以后使用 FileZilla 软件将正点原子提供的 uboot 源码拷贝到此目录中，正点原子提供的 uboot 源码已经放到了开发板光盘中，路径为：开发板光盘->1、例程源码->3、正点原子修改后的 Uboot 和 Linux-> uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2。将其拷贝到 Ubuntu 中新建的 alientek_uboot 文件夹下，完成以后如图 30.2.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ ls
uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 30.2.1 将 uboot 拷贝到 Ubuntu 中

使用如下命令对其进行解压缩：

```
tar -vxjf uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2
```

解压完成以后 alientek_uboot 文件夹内容如图 30.2.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ ls
api     config.mk  dts       Kconfig      Makefile  snapshot.commit
arch    configs    examples  lib          net       test
board   disk       fs        Licenses     post      tools
cmd     doc        include   MAINTAINERS  README    uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2
common  drivers    Kbuild    MAKEALL      scripts
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 30.2.2 解压后的 uboot

图 30.2.2 中除了 uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2 这个正点原子提供的 uboot 源码压缩包以外，其他的文件和文件夹都是解压出来的 uboot 源码。

1、512MB(DDR3)+8GB(EMMC)核心板

如果使用的是 512MB+8G 的 EMMC 核心板，使用如下命令来编译对应的 uboot：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
                                        mx6ull_14x14_ddr512_emmc_defconfig
make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j12
```

这三条命令中 ARCH=arm 设置目标为 arm 架构，CROSS_COMPILE 指定所使用的交叉编译器。第一条命令相当于“make distclean”，目的是清除工程，一般在第一次编译的时候最好清理一下工程。第二条指令相当于“make mx6ull_14x14_ddr512_emmc_defconfig”，用于配置 uboot，配置文件为 mx6ull_14x14_ddr512_emmc_defconfig。最后一条指令相当于 “make -j12”也就是使用 12 核来编译 uboot。当这三条命令执行完以后 uboot 也就编译成功了，如图 30.2.3 所示：

```
/mx6ullevk/imximage-ddr512.cfg.cfgtmp board/freescale/mx6ullevk/imximage-ddr512.cfg
  ./tools/mkimage -n board/freescale/mx6ullevk/imximage-ddr512.cfg.cfgtmp -T imximage -e 0x87
800000 -d u-boot.bin u-boot.imx
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:         DCD
Data Size:    425984 Bytes = 416.00 kB = 0.41 MB
Load Address: 877ff420
Entry Point:  87800000
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ ls
```

图 30.2.3 编译完成

编译完成以后的 alentek_uboot 文件夹内容如图 30.2.4 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ ls
api         doc         lib          scripts          u-boot.imx
arch        drivers     Licenses     snapshot.commit  uboot-imx-2016.03-2.1.0-g8b546e4.tar.bz2
board       dts         MAINTAINERS  System.map       u-boot.lds
cmd         examples    MAKEALL      test             u-boot.map
common      fs          Makefile     tools            u-boot-nodtb.bin
config.mk   include     net          u-boot           u-boot.srec
configs     Kbuild      post         u-boot.bin       u-boot.sym
disk        Kconfig     README       u-boot.cfg
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 30.2.4 编译后的 uboot 源码

可以看出，编译完成以后 uboot 源码多了一些文件，其中 u-boot.bin 就是编译出来的 uboot 二进制文件。uboot 是个裸机程序，因此需要在其前面加上头部(IVT、DCD 等数据)才能在 I.MX6U 上执行，图 30.2.4 中的 u-boot.imx 文件就是添加头部以后的 u-boot.bin，u-boot.imx 就是我们最终要烧写到开发板中的 uboot 镜像文件。

每次编译 uboot 都要输入一长串命令，为了简单起见，我们可以新建一个 shell 脚本文件，将这些命令写到 shell 脚本文件里面，然后每次只需要执行 shell 脚本即可完成编译工作。新建名为 mx6ull_alientek_emmc.sh 的 shell 脚本文件，然后在里面输入如下内容：

示例代码 30.2.1 mx6ull_alientek_emmc.sh 文件代码

```
1 #!/bin/bash
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
                          mx6ull_14x14_ddr512_emmc_defconfig
4 make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j12
```

第 1 行是 shell 脚本要求的，必须是“#!/bin/bash”或者“#!/bin/sh”。

第 2 行使用了 make 命令，用于清理工程，也就是每次在编译 uboot 之前都清理一下工程。这里的 make 命令带有三个参数，第一个是 ARCH，也就是指定架构，这里肯定是 arm；第二个参数 CROSS_COMPILE 用于指定编译器，只需要指明编译器前缀就行了，比如 arm-linux-gnueabihf-gcc 编译器的前缀就是“arm-linux-gnueabihf-”；最后一个参数 distclean 就是清除工程。

第 3 行也使用了 make 命令，用于配置 uboot。同样有三个参数，不同的是，最后一个参数是 mx6ull_alientek_emmc_defconfig。前面说了 uboot 是 bootloader 的一种，可以用来引导 Linux，但是 uboot 除了引导 Linux 以外还可以引导其它的系统，而且 uboot 还支持其它的架构和外设，比如 USB、网络、SD 卡等。这些都是可以配置的，需要什么功能就使能什么功能。所以在编译 uboot 之前，一定要根据自己的需求配置 uboot。mx6ull_alientek_emmc_defconfig 就是正点原子针对 I.MX6U-ALPHA 的 EMMC 核心板编写的配置文件，这个配置文件在 uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek /configs 目录中。在 uboot 中，通过“make xxx_defconfig”来配置 uboot，xxx_defconfig 就是不同板子的配置文件，这些配置文件都在 uboot/configs 目录中。

第 4 行有 4 个参数，用于编译 uboot，通过第 3 行配置好 uboot 以后就可以直接“make”编译 uboot 了。其中 V=1 用于设置编译过程的信息输出级别；-j 用于设置主机使用多少个核编译 uboot，设置的核越多，编译速度越快。-j16 表示使用 16 个核编译 uboot，具体设置多少个要根据自己的虚拟机或者电脑配置，如果你给 VMware 分配了 4 个核，那么最多只能使用-j4。

使用 chmod 命令给予 mx6ull_alientek_emmc.sh 文件可执行权限，然后就可以使用这个 shell 脚本文件来重新编译 uboot，命令如下：

```
./mx6ull_alientek_emmc.sh
```

1、256MB(DDR3)+256MB/512MB(NAND)核心板

如果用的 256MB+256MB/512MB 的 NAND 核心板，新建名为 mx6ull_alientek_nand.sh 的 shell 脚本文件，然后在里面输入如下内容：

示例代码 30.2.2 mx6ull_alientek_nand.sh 文件代码

```
1 #!/bin/bash
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
                           mx6ull_14x14_ddr256_nand_defconfig
4 make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j12
```

完成以后同样使用 chmod 指令给予 mx6ull_alientek_nand.sh 可执行权限，然后输入如下命令即可编译 NAND 版本的 uboot：

```
./mx6ull_alientek_nand.sh
```

mx6ull_alientek_nand.sh 和 mx6ull_alientek_emmc.sh 类似，只是 uboot 配置文件不同，这里就不详细介绍了。

30.3 U-Boot 烧写与启动

uboot 编译好以后就可以烧写到板子上使用了，这里我们跟前面裸机例程一样，将 uboot 烧写到 SD 卡中，然后通过 SD 卡来启动来运行 uboot。使用 imxdownload 软件烧写，命令如下：

```
chmod 777 imxdownload                //给予 imxdownload 可执行权限，一次即可
./imxdownload u-boot.bin    /dev/sdd
```

等待烧写完成，完成以后将 SD 卡插到 I.MX6U-ALPHA 开发板上，BOOT 设置从 SD 卡启动，使用 USB 线将 USB_TTL 和电脑连接，也就是将开发板的串口 1 连接到电脑上。打开 SecureCRT，设置好串口参数并打开，最后复位开发板。在 SecureCRT 上出现“Hit any key to stop autoboot: ”倒计时的时候按下键盘上的回车键，默认是 3 秒倒计时，在 3 秒倒计时结束以后如果没有按下回车键的话 uboot 就会使用默认参数来启动 Linux 内核了。如果在 3 秒倒计时结束之前按下回车键，那么就会进入 uboot 的命令行模式，如图 30.3.1 所示：

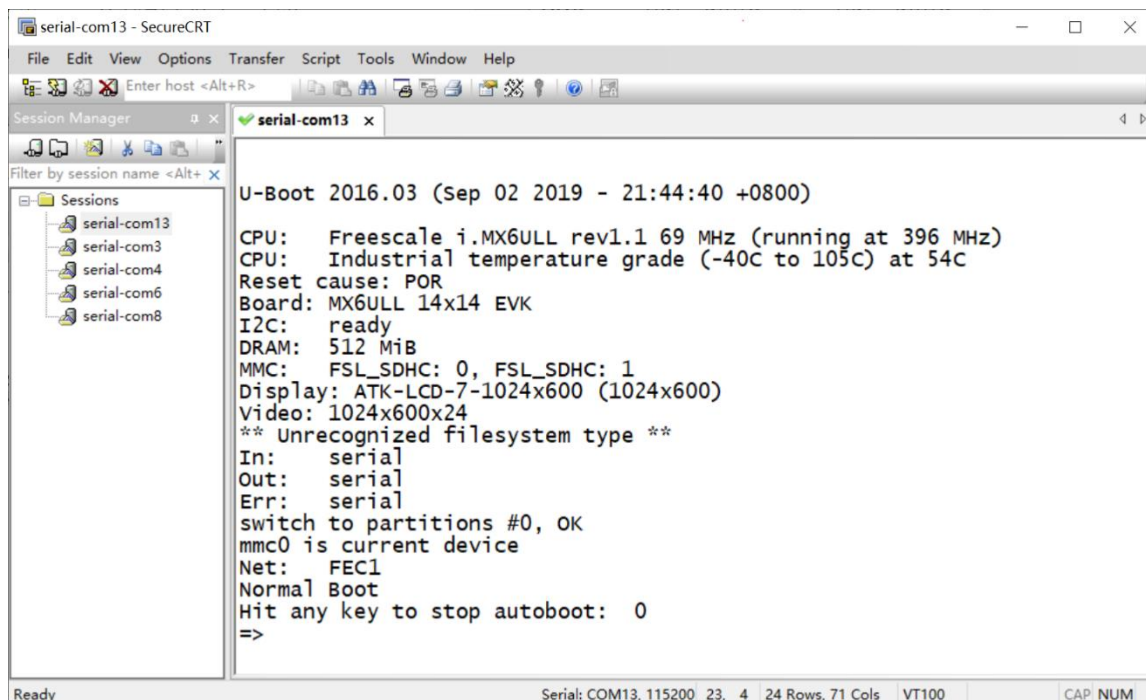


图 30.3.1 uboot 启动过程

从图 30.3.1 可以看出，当进入到 uboot 的命令行模式以后，左侧会出现一个“=>”标志。uboot 启动的时候会输出一些信息，这些信息如下所示：

示例代码 30.3.1 uboot 输出信息

```
1  U-Boot 2016.03 (Apr 12 2019 - 02:33:00 +0800)
2
3  CPU:   Freescale i.MX6ULL rev1.1 69 MHz (running at 396 MHz)
4  CPU:   Industrial temperature grade (-40C to 105C) at 46C
5  Reset cause: POR
6  Board: MX6ULL 14x14 EVK
7  I2C:   ready
8  DRAM:  512 MiB
9  MMC:   FSL_SDHC: 0, FSL_SDHC: 1
10 Display: ATK-LCD-7-1024x600 (1024x600)
11 Video: 1024x600x24
12 ** Unrecognized filesystem type **
13 In:    serial
14 Out:   serial
15 Err:   serial
16 switch to partitions #0, OK
17 mmc0 is current device
18 Net:   FEC1
19 Normal Boot
20 Hit any key to stop autoboot:  0
21 =>
```

第 1 行是 uboot 版本号和编译时间，可以看出，当前的 uboot 版本号是 2016.03，编译时间是 2019 年 4 月 12 日凌晨 2 点 33（没错！为了赶教程和例程，我这一年多以来基本每天晚上工作到凌晨两三点！看到这里一定要记得到论坛夸我一下!）。

第 3 和第 4 行是 CPU 信息，可以看出当前使用的 CPU 是飞思卡尔的 I.MX6ULL（I.MX 以前属于飞思卡尔，然而飞思卡尔被 NXP 收购了），如果使用 528MHz 的 I.MX6ULL，此处会显示主频为 528MHz。但是如果使用 800MHz 的 I.MX6ULL 的话此处会显示 69MHz，这个是 uboot 内部主频读取错误，但是不影响运行，可以不用管。不管是 528MHz 还是 800MHz 的 I.MX6ULL，此时都运行在 396MHz。这颗芯片是工业级的，可以工作在-40°C~105°C。

第 5 行是复位原因，当前的复位原因是 POR。I.MX6ULL 芯片上有个 POR_B 引脚，将这个引脚拉低即可复位 I.MX6ULL。

第 6 行是板子名字，当前的板子名字为“MX6ULL 14x14 EVK”。

第 7 行提示 I2C 准备就绪。

第 8 行提示当前板子的 DRAM（内存）为 512MB，如果是 NAND 版本的话内存为 256MB。

第 9 行提示当前有两个 MMC/SD 卡控制器：FSL_SDHC(0)和 FSL_SDHC(1)。I.MX6ULL 支持两个 MMC/SD，正点原子的 I.MX6ULL EMMC 核心板上 FSL_SDHC(0)接的 EMMC，FSL_SDHC(1)接的 SD(TF)卡。

第 10 和第 11 行是 LCD 型号，当前的 LCD 型号是 ATK-LCD-7-1024x600 (1024x600)，分辨率为 1024x600，格式为 RGB888(24 位)。

第 13~15 是标准输入、标准输出和标准错误所使用的终端，这里都使用串口(serial)作为终端。

第 16 和 17 行是切换到 emmc 的第 0 个分区上，因为当前的 uboot 是 emmc 版本的，也就是从 emmc 启动的。我们只是为了方便将其烧写到了 SD 卡上，但是它的“内心”还是 EMMC 的。所以 uboot 启动以后会将 emmc 作为默认存储器，当然了，你也可以将 SD 卡作为 uboot 的存储器，这个我们后面会讲解怎么做。

第 18 行是网口信息，提示我们当前使用的 FEC1 这个网口，I.MX6ULL 支持两个网口。

第 19 行提示正常启动，也就是说 uboot 要从 emmc 里面读取环境变量和参数信息启动 Linux 内核了。

第 20 行是倒计时提示，默认倒计时 3 秒，倒计时结束之前按下回车键就会进入 Linux 命令行模式。如果在倒计时结束以后没有按下回车键，那么 Linux 内核就会启动，Linux 内核一旦启动，uboot 就会寿终正寝。

这个就是 uboot 默认输出信息的含义，NAND 版本的 uboot 也是类似的，只是 NAND 版本的就没有 EMMC/SD 相关信息了，取而代之的就是 NAND 的信息，比如 NAND 容量大小信息。

uboot 是来干活的，我们现在已经进入 uboot 的命令行模式了，进入命令行模式以后就可以给 uboot 发号施令了。当然了，不能随便发号施令，得看看 uboot 支持哪些命令，然后使用这些 uboot 所支持的命令来做一些工作。下一节就讲解 uboot 命令的使用。

30.4 U-Boot 命令使用

进入 uboot 的命令行模式以后输入“help”或者“？”，然后按下回车即可查看当前 uboot 所支持的命令，如图 30.4.1 所示：

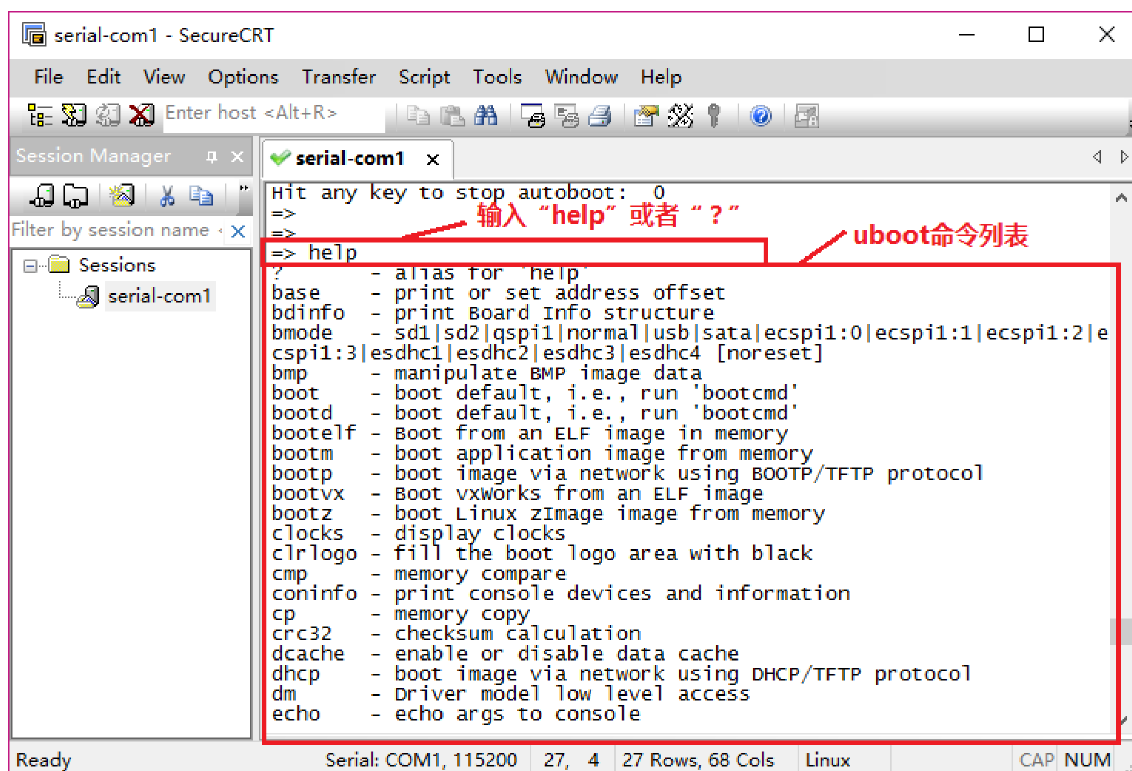


图 30.4.1 uboot 命令列表

图 30.4.1 中只是 uboot 的一部分命令，具体的命令列表以实际为准。图 30.4.1 中的命令并不是 uboot 所支持的所有命令，前面说过 uboot 是可配置的，需要什么命令就使能什么命令。所

以图 30.4.1 中的命令是正点原子提供的 uboot 中使能的命令，uboot 支持的命令还有很多，而且也可以在 uboot 中自定义命令。这些命令后面都跟有命令说明，用于描述此命令的作用，但是命令具体怎么用呢？我们输入“help(或?) 命令名”既可以查看命令的详细用法，以“bootz”这个命令为例，我们输入如下命令即可查看“bootz”这个命令的用法：

```
? bootz 或 help bootz
```

结果如图 30.4.2 所示：

```
=> ? bootz
bootz - boot Linux zImage image from memory

Usage:
bootz [addr [initrd[:size]] [fdt]]
    - boot Linux zImage stored in memory
        The argument 'initrd' is optional and specifies the address
        of the initrd in memory. The optional argument ':size' allows
        specifying the size of RAW initrd.
        When booting a Linux kernel which requires a flat device-tree
        a third argument is required which is the address of the
        device-tree blob. To boot that kernel without an initrd image,
        use a '-' for the second argument. If you do not pass a third
        a bd_info struct will be passed instead

=>
Serial: COM1, 115200   36, 4   36 Rows, 72 Cols   Linux   CAP NUM
```

图 30.4.2 bootz 命令使用说明

图 30.4.2 中就详细的列出了“bootz”这个命令的详细，其它的命令也可以使用此方法查询具体的使用方法。接下来我们学习一下一些常用的 uboot 命令。

30.4.1 信息查询命令

常用的和信息查询有关的命令有 3 个：bdinfo、printenv 和 version。先来看一下 bdinfo 命令，此命令用于查看板子信息，直接输入“bdinfo”即可，结果如图 30.4.1.1 所示：

```
=> bdinfo
arch_number = 0x00000000
boot_params = 0x80000100
DRAM bank   = 0x00000000
-> start    = 0x80000000
-> size     = 0x20000000
eth0name    = FEC1
ethaddr     = (not set)
current eth = FEC1
ip_addr     = <NULL>
baudrate    = 115200 bps
TLB addr    = 0x9FFF0000
relocaddr   = 0x9FF47000
reloc off   = 0x18747000
irq_sp      = 0x9EF44EA0
sp start    = 0x9EF44E90
FB base     = 0x00000000
=>
```

图 30.4.1.1 bdinfo 命令

从图 30.4.1.1 中可以得出 DRAM 的其实地址和大小、启动参数保存起始地址、波特率、sp(堆栈指针)起始地址等信息。

命令“printenv”用于输出环境变量信息，uboot 也支持 TAB 键自动补全功能，输入“print”然后按下 TAB 键就会自动补全命令，直接输入“print”也可以。输入“print”，然后按下回车键，环境变量如图 30.4.1.2 所示：

```
=> print
baudrate=115200
board_name=EVK
board_rev=14X14
boot_fdt=try
bootcmd=run findfdt;mmc dev ${mmcdev};mmc dev ${mmcdev}; if mmc rescan; then if run loadbootscript;
then run bootscript; else if run loadimage; then run mmcboot; else run netboot; fi; fi; else run net
boot; fi
bootcmd_mfg=run mfgtool_args;bootz ${loadaddr} ${initrd_addr} ${fdt_addr};
bootdelay=3
bootscript=echo Running bootscript from mmc ...; source
console=ttymxc0
ethact=FEC1
ethprime=FEC
fdt_addr=0x83000000
fdt_file=undefined
fdt_high=0xffffffff
findfdt=if test $fdt_file = undefined; then if test $board_name = EVK && test $board_rev = 9X9; then
 setenv fdt_file imx6ull-9x9-evk.dtb; fi; if test $board_name = EVK && test $board_rev = 14X14; then
 setenv fdt_file imx6ull-14x14-evk.dtb; fi; if test $fdt_file = undefined; then echo WARNING: Could
not determine dtb to use; fi; fi;
image=zImage
initrd_addr=0x83800000
initrd_high=0xffffffff
ip_dyn=yes
loadaddr=0x80800000
loadbootscript=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${script};
loadfdt=fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr} ${fdt_file}
loadimage=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${image}
mfgtool_args=setenv bootargs console=${console},${baudrate} rdinit=/linuxrc g_mass_storage.stall=0 g
_mass_storage.removable=1 g_mass_storage.file=/fat g_mass_storage.ro=1 g_mass_storage.idVendor=0x066
F g_mass_storage.idProduct=0x37FF g_mass_storage.iSerialNumber="" clk_ignore_unused
mmcargs=setenv bootargs console=${console},${baudrate} root=${mmcroot}
mmcautodetect=yes
mmcboot=echo Booting from mmc ...; run mmcargs; if test ${boot_fdt} = yes || test ${boot_fdt} = try;
 then if run loadfdt; then bootz ${loadaddr} - ${fdt_addr}; else if test ${boot_fdt} = try; then boo
tz; else echo WARN: Cannot load the DT; fi; fi; else bootz; fi;
mmcdev=1
mmcpart=1
mmcroot=/dev/mmcblk1p2 rootwait rw
netargs=setenv bootargs console=${console},${baudrate} root=/dev/nfs ip=dhcp nfsroot=${serverip}:${n
fsroot},v3,tcp
netboot=echo Booting from net ...; run netargs; if test ${ip_dyn} = yes; then setenv get_cmd dhcp; e
lse setenv get_cmd tftp; fi; ${get_cmd} ${image}; if test ${boot_fdt} = yes || test ${boot_fdt} = tr
y; then if ${get_cmd} ${fdt_addr} ${fdt_file}; then bootz ${loadaddr} - ${fdt_addr}; else if test ${
boot_fdt} = try; then bootz; else echo WARN: Cannot load the DT; fi; fi; else bootz; fi;
panel=TFT43AB
script=boot.scr

Environment size: 2431/8188 bytes
=>
```

图 30.4.1.2 printenv 命令结果

在图 30.4.1.2 中有很多的环境变量，比如 baudrate、board_name、board_rec、boot_fdt、bootcmd 等等。uboot 中的环境变量都是字符串，既然叫做环境变量，那么它的作用就和“变量”一样。比如 bootdelay 这个环境变量就表示 uboot 启动延时时间，默认 bootdelay=3，也就默认延时 3 秒。前面说的 3 秒倒计时就是由 bootdelay 定义的，如果将 bootdelay 改为 5 的话就会倒计时 5s 了。uboot 中的环境变量是可以修改的，有专门的命令来修改环境变量的值，稍后我们会讲解。

命令 version 用于查看 uboot 的版本号，输入“version”，uboot 版本号如图 30.4.1.3 所示：

```
=> version

U-Boot 2016.03 (Apr 12 2019 - 02:33:00 +0800)
arm-linux-gnueabihf-gcc (Linaro GCC 4.9-2017.01) 4.9.4
GNU ld (Linaro_Binutils-2017.01) 2.24.0.20141017 Linaro 2014_11-3-git
=>
```

图 30.4.1.3 version 命令结果

从图 30.4.1.3 可以看出，当前 uboot 版本号为 2016.03，2019 年 4 月 12 日编译的，编译器为 arm-linux-gnueabihf-gcc 等信息。

30.4.2 环境变量操作命令

1、修改环境变量

环境变量的操作涉及到两个命令：setenv 和 saveenv，命令 setenv 用于设置或者修改环境变量的值。命令 saveenv 用于保存修改后的环境变量，一般环境变量是存放在外部 flash 中的，

uboot 启动的时候会将环境变量从 flash 读取到 DRAM 中。所以使用命令 setenv 修改的是 DRAM 中的环境变量值，修改以后要使用 saveenv 命令将修改后的环境变量保存到 flash 中，否则的话 uboot 下一次重启会继续使用以前的环境变量值。

命令 saveenv 使用起来很简单，格式为：

```
saveenv 命令 值
```

或

```
saveenv 命令 ‘值 1 值 2 值 3’
```

比如我们要将环境变量 bootdelay 该为 5，就可以使用如下所示命令：

```
setenv bootdelay 5
saveenv
```

上述命令执行过程如图 30.4.2.1 所示：

```
=> setenv bootdelay 5
=> saveenv
Saving Environment to MMC...
Writing to MMC(1)... done
=>
```

图 30.4.2.1 环境变量修改

在图 30.4.2.1 中，当我们使用命令 saveenv 保存修改后的环境变量的话会有保存过程提示信息，根据提示可以看出环境变量保存到了 MMC(1)中，也就是 EMMC 中。因为我用的 EMMC 版本的核心板，所以会保存到 MMC(1)中，如果是 NAND 版本核心板的话就会提示保存到 NAND 中。

修改 bootdelay 以后，重启开发板，uboot 就是变为 5 秒倒计时，如图 30.4.2.2 所示：

```
U-Boot 2016.03 (Apr 12 2019 - 02:33:00 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 52C
Reset cause: WDOG
Board: MX6ULL ALIENTEK EMMC
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
unsupported panel TFT43AB
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc1(part 0) is current device
Net:   FEC1
Error: FEC1 address not set.

Normal Boot
Hit any key to stop autoboot:  5
```

图 30.4.2.2 5 秒倒计时

有时候我们修改的环境变量值可能会有空格，比如 bootcmd、bootargs 等，这个时候环境变量值就得用单引号括起来，比如下面修改环境变量 bootcmd 的值：

```
setenv bootcmd 'console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw'
saveenv
```

上面命令设置 bootcmd 的值为“console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw”，其中“console=ttymxc0,115200”、“root=/dev/mmcblk1p2”、“rootwait”和“rw”相当于四组“值”，这四组“值”之间用空格隔开，所以需要使用单引号‘’将其括起来，表示这四组“值”都属于环境变量 bootcmd。

2、新建环境变量

命令 setenv 也可以用于新建命令，用法就是修改环境变量一样，比如我们新建一个环境变量 author，author 的值为我的名字拼音：zuozhongkai，那么就可以使用如下命令：

```
setenv author zuozhongkai
saveenv
```

新建命令 author 完成以后重启 uboot，然后使用命令 printenv 查看当前环境变量，如图 30.4.2.3 所示：

```
=> print
author=zuozhongkai
baudrate=115200
board_name=EVK
board_rev=14X14
boot_fdt=try
```

图 30.4.2.3 环境变量

从图 30.4.2.3 可以看到新建的环境变量：author，其值为：zuozhongkai。

3、删除环境变量

既然可以新建环境变量，那么就可以删除环境变量，删除环境变量也是使用命令 setenv，要删除一个环境变量只要给这个环境变量赋空值即可，比如我们删除掉上面新建的 author 这个环境变量，命令如下：

```
setenv author
saveenv
```

上面命令中通过 setenv 给 author 赋空值，也就是什么都不写来删除环境变量 author。重启 uboot 就会发现环境变量 author 没有了。

30.4.3 内存操作命令

内存操作命令就是用于直接对 DRAM 进行读写操作的，常用的内存操作命令有 md、nm、mm、mw、cp 和 cmp。我们依次来看一下这些命令都是做什么的。

1、md 命令

md 命令用于显示内存值，格式如下：

```
md[.b, .w, .l] address [# of objects]
```

命令中的[.b .w .l]对应 byte、word 和 long，也就是分别以 1 个字节、2 个字节、4 个字节来显示内存值。address 就是要查看的内存起始地址，[# of objects]表示要查看的数据长度，这个数据长度单位不是字节，而是跟你所选择的显示格式有关。比如你设置要查看的内存长度问为 20(十六进制为 0x14)，如果显示格式为.b 的话那就表示 20 个字节；如果显示格式为.w 的话就表示 20 个 word，也就是 20*2=40 个字节；如果显示格式为.l 的话就表示 20 个 long，也就是 20*4=80 个字节。另外要注意：

uboot 命令中的数字都是十六进制的！不是十进制的！
uboot 命令中的数字都是十六进制的！不是十进制的！
uboot 命令中的数字都是十六进制的！不是十进制的！

比如你想查看以 0X80000000 开始的 20 个字节的内存值，显示格式为.b 的话，应该使用如下所示命令：

```
md.b 80000000 14
```

而不是：

```
md.b 80000000 20
```

上面说了，uboot 命令里面的数字都是十六进制的，所以可以不用写“0x”前缀，十进制的 20 其十六进制为 0x14，所以命令 md 后面的个数应该是 14，如果写成 20 的话就表示查看 32(十六进制为 0x20)个字节的数据。分析下面三个命令的区别：

```
md.b 80000000 10
md.w 80000000 10
md.l 80000000 10
```

上面这三个命令都是查看以 0X80000000 为起始地址的内存数据，第一个命令以.b 格式显示，长度为 0x10，也就是 16 个字节；第二个命令以.w 格式显示，长度为 0x10，也就是 16*2=32 个字节；最后一个命令以.l 格式显示，长度也是 0x10，也就是 16*4=64 个字节。这三个命令的执行结果如图 30.4.3.1 所示：

```
=> md.b 80000000 10
80000000: ff ff ff ff ff af ff ff ff ff f9 ff ff fe bf ff    ................
=> md.w 80000000 10
80000000: ffff ffff afff ffff ffff fff9 feff ffbf    ................
80000010: feef ffff ffff 3fff ffff fbff ffff fbfb    .......?........
=> md.l 80000000 10
80000000: ffffffff ffffafff fff9ffff ffbffeff    ................
80000010: fffffeef 3fffffff fbffffff fbfbffff    .......?........
80000020: ff7fffff fdfffdff efffffff fff7ff7e    ............~...
80000030: effbfbff ff7fffd7 fffdffff fefbffff    ................
=>
```

图 30.4.3.1 md 命令使用示例

2、nm 命令

nm 命令用于修改指定地址的内存值，命令格式如下：

```
nm [.b, .w, .l] address
```

nm 命令同样可以以.b、.w 和.l 来指定操作格式，比如现在以.l 格式修改 0x80000000 地址的数据为 0x12345678。输入命令：

```
nm.l 80000000
```

输入上述命令以后如图 30.4.3.2 所示：

```
=> nm.l 80000000
80000000: ffffff00 ?
```

图 30.4.3.2 nm 命令

在图 30.4.3.2 中，80000000 表示现在要修改的内存地址，ffffff00 表示地址 0x80000000 现在的数据，？后面就可以输入要修改后的数据 0x12345678，输入完成以后按下回车，然后再输入‘q’即可退出，如图 30.4.3.3 所示：

```
=> nm.l 80000000
80000000: ffffff00 ? 12345678
80000000: 12345678 ? q
=>
```

图 30.4.3.3 修改内存数据

修改完成以后在使用命令 md 来查看一下有没有修改成功，如图 30.4.3.4 所示：

```
=> md.l 80000000 1
80000000: 12345678                                    xV4.
=>
```

图 30.4.3.4 查看修改后的值

从图 30.4.3.4 可以看出，此时地址 0X80000000 的值变为了 0x12345678。

3、mm 命令

mm 命令也是修改指定地址内存值的，使用 mm 修改内存值的时候地址会自增，而使用命令 nm 的话地址不会自增。比如以.l 格式修改从地址 0x80000000 开始的连续 3 个内存块(3*4=12 个字节)的数据为 0X05050505，操作如图 30.4.3.5 所示：

```
=> mm.l 80000000
80000000: 12345678 ? 05050505
80000004: ffffafff ? 05050505
80000008: fff9ffff ? 05050505
8000000c: ffbffeff ? q
=>
```

图 30.4.3.5 命令 mm

从图 30.4.3.5 可以看出，修改了地址 0X80000000、0X80000004 和 0X8000000C 的内容为 0x05050505。使用命令 md 查看修改后的值，结果如图 30.4.3.6 所示：

```
=> md.l 80000000 3
80000000: 05050505 05050505 05050505            ............
=>
```

图 30.4.3.6 查看修改后的内存数据

从图 30.4.3.6 可以看出内存数据修改成功。

4、mw 命令

命令 mw 用于使用一个指定的数据填充一段内存，命令格式如下：

```
mw [.b, .w, .l] address value [count]
```

mw 命令同样可以以.b、.w 和.l 来指定操作格式，address 表示要填充的内存起始地址，value 为要填充的数据，count 是填充的长度。比如使用.l 格式将以 0X80000000 为起始地址的 0x10 个内存块(0x10 * 4=64 字节)填充为 0X0A0A0A0A，命令如下：

```
mw.l 80000000 0A0A0A0A 10
```

然后使用命令 md 来查看，如图 30.4.3.7 所示：

```
=> mw.l 80000000 0A0A0A0A 10
=> md.l 80000000 10
80000000: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000010: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000020: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000030: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
=>
```

图 30.4.3.7 查看修改后的内存数据

从图 30.4.3.7 可以看出内存数据修改成功。

5、cp 命令

cp 是数据拷贝命令，用于将 DRAM 中的数据从一段内存拷贝到另一段内存中，或者把 Nor Flash 中的数据拷贝到 DRAM 中。命令格式如下：

```
cp [.b, .w, .l] source target count
```

cp 命令同样可以以.b、.w 和.l 来指定操作格式，source 为源地址，target 为目的地址，count 为拷贝的长度。我们使用.l 格式将 0x80000000 处的地址拷贝到 0X80000100 处，长度为 0x10 个内存块(0x10 * 4=64 个字节)，命令如下所示：

```
cp.l 80000000 80000100 10
```

结果如图 30.4.3.8 所示：

```
=> md.l 80000000 10
80000000: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000010: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000020: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000030: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
=> md.l 80000100 10
80000100: ffff7fff ffffffdf fbffffff ffffffdf    ................
80000110: 7fffefff fdffffff ffeff7ff fb7dd7ff    ..............}.
80000120: bfffedff efdff7ed efbf7fff ffffffff    ................
80000130: fffffffb fbffffff dfdefcff fffbffdf    ................
=> cp.l 80000000 80000100 10
=> md.l 80000100 10
80000100: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000110: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000120: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
80000130: 0a0a0a0a 0a0a0a0a 0a0a0a0a 0a0a0a0a    ................
=>
```

图 30.4.3.8 cp 命令操作结果

在图 30.4.3.8 中，先使用 md.l 命令打印出地址 0x80000000 和 0x80000100 处的数据，然后使用命令 cp.l 将 0x80000100 处的数据拷贝到 0x80000100 处。最后使用命令 md.l 查看 0x80000100 处的数据有没有变化，检查拷贝是否成功。

6、cmp 命令

cmp 是比较命令，用于比较两段内存的数据是否相等，命令格式如下：

```
cmp [.b, .w, .l] addr1 addr2 count
```

cmp 命令同样可以以.b、.w 和.l 来指定操作格式，addr1 为第一段内存首地址，addr2 为第二段内存首地址，count 为要比较的长度。我们使用.l 格式来比较 0x80000000 和 0X80000100 这两个地址数据是否相等，比较长度为 0x10 个内存块(16 * 4=64 个字节)，命令如下所示：

```
cmp.l 80000000 80000100 10
```

结果如图 30.4.3.9 所示：

```
=> cmp.l 80000000 80000100 10
Total of 16 word(s) were the same
=>
```

图 30.4.3.9 cmp 命令比较结果

从图 30.4.3.9 可以看出两段内存的数据相等。我们再随便挑两段内存比较一下，比如地址 0x80002000 和 0x800003000，长度为 0X10，比较结果如图 30.4.3.10 所示：

```
=> cmp.l 80002000 80003000 10
word at 0x80002000 (0xffff7ff7) != word at 0x80003000 (0xfffff7ff)
Total of 0 word(s) were the same
=>
```

图 30.4.3.10 cmp 命令比较结果

从图 30.4.3.10 可以看出，0x80002000 处的数据和 0x80003000 处的数据就不一样。

30.4.4 网络操作命令

uboot 是支持网络的，我们在移植 uboot 的时候一般都要调通网络功能，因为在移植 linux kernel 的时候需要使用到 uboot 的网络功能做调试。uboot 支持大量的网络相关命令，比如 dhcp、ping、nfs 和 tftpboot，我们接下来依次学习一下这几个和网络有关的命令。

在使用 uboot 的网络功能之前先用网线将开发板的 ENET2 接口和电脑或者路由器连接起来，I.MX6U-ALPHA 开发板有两个网口：ENET1 和 ENET2，一定要连接 ENET2，不能连接错了，ENET2 接口如图 30.4.4.1 所示。

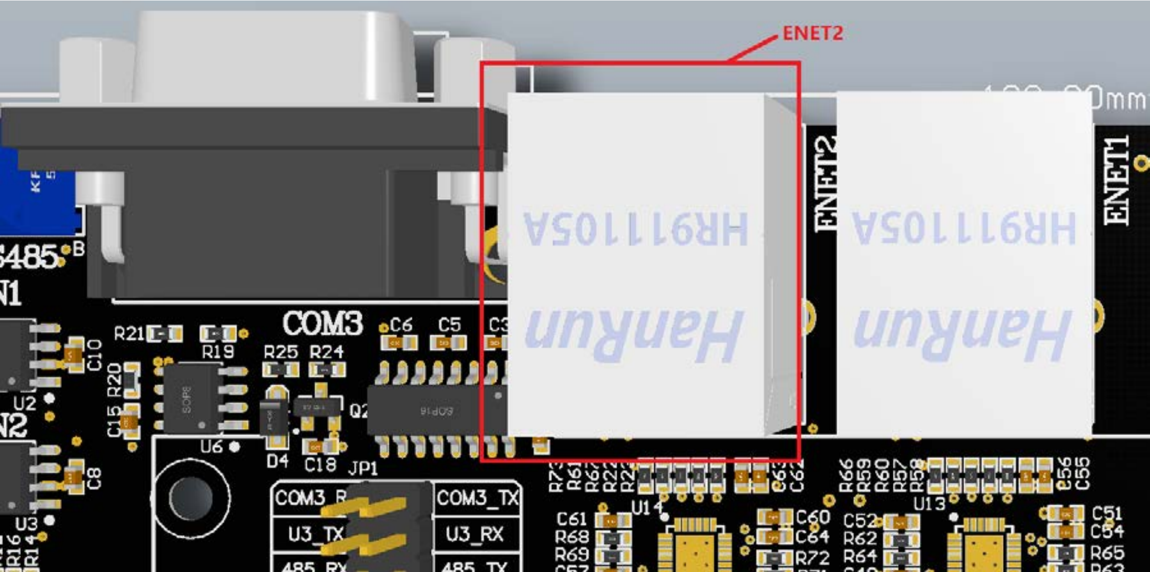


图 30.4.4.1 ENET2 网络接口

建议开发板和主机 PC 都连接到同一个路由器上！最后设置表 30.4.4.1 中所示的几个环境变量。

环境变量	描述
ipaddr	开发板 ip 地址，可以不设置，使用 dhcp 命令来从路由器获取 IP 地址。
ethaddr	开发板的 MAC 地址，一定要设置。
gatewayip	网关地址。
netmask	子网掩码。
serverip	服务器 IP 地址，也就是 Ubuntu 主机 IP 地址，用于调试代码。

表 30.4.4.1 网络相关环境变量

表 30.4.4.1 中环境变量设置命令如下所示：

```
setenv ipaddr 192.168.1.50
setenv ethaddr 00:04:9f:04:d2:35
setenv gatewayip 192.168.1.1
setenv netmask 255.255.255.0
setenv serverip 192.168.1.250
saveenv
```

注意，网络地址环境变量的设置要根据自己的实际情况，确保 Ubuntu 主机和开发板的 IP 地址在同一个网段内，比如我现在的开发板和电脑都在 192.168.1.0 这个网段内，所以设置开发板的 IP 地址为 192.168.1.50，我的 Ubuntu 主机的地址为 192.168.1.250，因此 serverip 就是 192.168.1.250。ethaddr 为网络 MAC 地址，是一个 48bit 的地址，如果在同一个网段内有多个开发板的话一定要保证每个开发板的 ethaddr 是不同的，否则通信会有问题！设置好网络相关的环境变量以后就可以使用网络相关命令了。

1、ping 命令

开发板的网络能否使用，是否可以和服务器(Ubuntu 主机)进行通信，通过 ping 命令就可以验证，直接 ping 服务器的 IP 地址即可，比如我的服务器 IP 地址为 192.168.1.250，命令如下：

```
ping 192.168.1.250
```

结果如图 30.4.4.2 所示：

```
=> ping 192.168.1.250
Using FEC1 device
host 192.168.1.250 is alive
=>
```

图 30.4.4.2 ping 命令

从图 30.4.4.2 可以看出，192.168.1.250 这个主机存在，说明 ping 成功，uboot 的网络工作正常。

2、dhcp 命令

dhcp 用于从路由器获取 IP 地址，前提得开发板连接到路由器上的，如果开发板是和电脑直连的，那么 dhcp 命令就会失效。直接输入 dhcp 命令即可通过路由器获取到 IP 地址，如图 30.4.4.3 所示：

```
=> dhcp
BOOTP broadcast 1
*** Unhandled DHCP Option in OFFER/ACK: 50
*** Unhandled DHCP Option in OFFER/ACK: 50
DHCP client bound to address 192.168.1.50 (10 ms)
*** Warning: no boot file name; using 'C0A80132.img'
Using FEC1 device
TFTP from server 192.168.1.1; our IP address is 192.168.1.50
Filename 'C0A80132.img'.
Load address: 0x80800000
Loading: T T T T T T
```

图 30.4.4.3 dhcp 命令

从图 30.4.4.3 可以看出，开发板通过 dhcp 获取到的 IP 地址为 192.168.1.50，和我们手动设置的一样，这很正常。同时在图 30.4.4.3 中可以看到“warning：no boot file name;”、“TFTP from server 192.168.1.1”这样的字样。这是因为 DHCP 不单单是获取 IP 地址，其还会通过 TFTP 来启动 linux 内核，输入“? dhcp”即可查看 dhcp 命令详细的信息，如图 30.4.4.4 所示：

```
=> ? dhcp
dhcp - boot image via network using DHCP/TFTP protocol

Usage:
dhcp [loadAddress] [[hostIPaddr:]bootfilename]
=>
```

图 30.4.4.4 dhcp 命令使用查询

3、nfs 命令

nfs 也就是网络文件系统，通过 nfs 可以在计算机之间通过网络来分享资源，比如我们将 linux 镜像和设备树文件放到 Ubuntu 中，然后在 uboot 中使用 nfs 命令将 Ubuntu 中的 linux 镜像和设备树下载到开发板的 DRAM 中。这样做的目的是为了方便调试 linux 镜像和设备树，也就是网络调试，通过网络调试是 Linux 开发中最常用的调试方法。原因是嵌入式 linux 开发不像单片机开发，可以直接通过 JLINK 或 STLink 等仿真器将代码直接烧写到单片机内部的 flash 中，嵌入式 Linux 通常是烧写到 EMMC、NAND Flash、SPI Flash 等外置 flash 中，但是嵌入式 Linux 开发也没有 MDK，IAR 这样的 IDE，更没有烧写算法，因此不可能通过点击一个“download”按钮就将固件烧写到外部 flash 中。虽然半导体厂商一般都会提供一个烧写固件的软件，但是这个软件使用起来比较复杂，这个烧写软件一般用于量产的。其远没有 MDK、IAR 的一键下载方便，在 Linux 内核调试阶段，如果用这个烧写软件的话将会非常浪费时间，而这个时候网络调试的优势就显现出来了，可以通过网络将编译好的 linux 镜像和设备树文件下载到 DRAM 中，然后就可以直接运行。

我们一般使用 uboot 中的 nfs 命令将 Ubuntu 中的文件下载到开发板的 DRAM 中，在使用之前需要开启 Ubuntu 主机的 NFS 服务，并且要新建一个 NFS 使用的目录，以后所有要通过 NFS 访问的文件都需要放到这个 NFS 目录中。Ubuntu 的 NFS 服务开启我们在 4.2.1 小节已经详细讲解过了，包括 NFS 文件目录的创建，如果忘记的话可以去查看一下 4.2.1 小节。我设置的/home/zuozhongkai/linux/nfs 这个目录为我的 NFS 文件目录。uboot 中的 nfs 命令格式如下所示：

```
nfs [loadAddress] [[hostIPaddr:]bootfilename]
```

loadAddress 是要保存的 DRAM 地址，[[hostIPaddr:]bootfilename]是要下载的文件地址。这里我们将正点原子官方编译出来的 Linux 镜像文件 zImage 下载到开发板 DRAM 的 0x80800000 这个地址处。正点原子编译出来的 zImage 文件已经放到了开发板光盘中，路径为：8、开发板系统镜像->zImage。将文件 zImage 通过 FileZilla 发送到 Ubuntu 中的 NFS 目录下，比如我的就是放到/home/zuozhongkai/linux/nfs 这个目录下，完成以后的 NFS 目录如图 30.4.4.5 所示：

```
zuozhongkai@ubuntu:~/linux/nfs$ ls -l
总用量 3598528
drwxr-xr-x 20 zuozhongkai zuozhongkai       4096 Mar 26 12:23 debian_rootfs
-rw-r--r--  1 zuozhongkai zuozhongkai      37162 Apr 15 12:39 imx6ul-14x14-evk.dtb
-rw-r--r--  1 zuozhongkai zuozhongkai      36626 Mar 26 12:09 imx6ull-14x14-evk.dtb
drwxr-xr-x 30 zuozhongkai zuozhongkai       4096 Apr 16 18:21 imx_rootfs
drwxr-xr-x  3 zuozhongkai zuozhongkai       4096 Mar 26 12:25 imx_rootfs_alsa
-rw-r--r--  1 zuozhongkai zuozhongkai  526147441 Mar 26 12:11 imx_rootfs_alsa.tar.bz2
-rw-r--r--  1 zuozhongkai zuozhongkai  529069027 Mar 26 12:10 imx_rootfs.tar.bz2
-rw-------  1 zuozhongkai zuozhongkai  625374408 Mar 26 12:11 linaro-jessie-alip-20160428-22.tar.gz
drwxr-xr-x 19 zuozhongkai zuozhongkai       4096 Mar 26 12:24 nxp_rootfs
-rw-r--r--  1 zuozhongkai zuozhongkai 1986363826 Mar 26 12:10 rootfs_nogpu.tar.bz2
drwxr-xr-x 23 zuozhongkai zuozhongkai       4096 Mar 26 12:21 ubuntu_rootfs
-rw-------  1 zuozhongkai zuozhongkai    6071136 Apr 18 12:18 zImage
-rwxr-xr-x  1 zuozhongkai zuozhongkai    6071136 Apr 17 18:17 zImage_imx6ul
-rwxr-xr-x  1 zuozhongkai zuozhongkai    5662880 Mar 26 12:09 zImage_imx6ull
```

zImage镜像文件

图 30.4.4.5 NFS 目录中的 zImage 文件

准备好以后就可以使用 nfs 命令来将 zImage 下载到开发板 DRAM 的 0X80800000 地址处，命令如下：

```
nfs 80800000 192.168.1.250:/home/zuozhongkai/linux/nfs/zImage
```

命令中的“80800000”表示 zImage 保存地址，“192.168.1.250:/home/zuozhongkai/linux/nfs/zImage”表示 zImage 在 192.168.1.250 这个主机中，路径为/home/zuozhongkai/linux/nfs/zImage。下载过程如图 30.4.4.6 所示：

```
=> nfs 80800000 192.168.1.250:/home/zuozhongkai/linux/nfs/zImage
Using FEC1 device
File transfer via NFS from server 192.168.1.250; our IP address is 192.168.1.5
Filename '/home/zuozhongkai/linux/nfs/zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ################
done
Bytes transferred = 6071136 (5ca360 hex)
=>
```

图 30.4.4.6 nfs 命令下载 zImage 过程

在图 30.4.4.6 中会以“#”提示下载过程，下载完成以后会提示下载的数据大小，这里下载的 6071136 字节，而 zImage 的大小就是 6071136 字节，如图 30.4.4.7 所示：

```
zuozhongkai@ubuntu:~/linux/nfs$ ls zImage -l
-rw------- 1 zuozhongkai zuozhongkai 6071136 Apr 18 12:18 zImage
zuozhongkai@ubuntu:~/linux/nfs$
```

图 30.4.4.7 zImage 大小

下载完成以后查看 0x80800000 地址处的数据，使用命令 md.b 来查看前 100 个字节的数据，如图 30.4.4.8 所示：

```
=> md.b 80800000 100
80800000: 00 00 a0 e1 00 00 a0 e1 00 00 a0 e1 00 00 a0 e1    ................
80800010: 00 00 a0 e1 00 00 a0 e1 00 00 a0 e1 00 00 a0 e1    ................
80800020: 03 00 00 ea 18 28 6f 01 00 00 00 00 60 a3 5c 00    .....(o.....`.\.
80800030: 01 02 03 04 00 90 0f e1 e8 04 00 eb 01 70 a0 e1    ..............p..
80800040: 02 80 a0 e1 00 20 0f e1 03 00 12 e3 01 00 00 1a    ..... ..........
80800050: 17 00 a0 e3 56 34 12 ef 00 00 0f e1 1a 00 20 e2    ....V4........ .
80800060: 1f 00 10 e3 1f 00 c0 e3 d3 00 80 e3 04 00 00 1a    ................
80800070: 01 0c 80 e3 0c e0 8f e2 00 f0 6f e1 0e f3 2e e1    ...........o.....
80800080: 6e 00 60 e1 00 f0 21 e1 09 f0 6f e1 00 00 00 00    n.`...!...o.....
80800090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
808000a0: 0f 40 a0 e1 3e 43 04 e2 02 49 84 e2 0f 00 a0 e1    .@..>C...I......
808000b0: 04 00 50 e1 ac 01 9f 35 0f 00 80 30 00 00 54 31    ..P....5...0..T1
808000c0: 01 40 84 33 6d 00 00 2b 5e 0f 8f e2 4e 1c 90 e8    .@.3m..+^...N...
808000d0: 1c d0 90 e5 01 00 40 e0 00 60 86 e0 00 a0 8a e0    ......@..`......
808000e0: 00 90 da e5 01 e0 da e5 0e 94 89 e1 02 e0 da e5    ................
808000f0: 03 a0 da e5 0e 98 89 e1 0a 9c 89 e1 00 d0 8d e0    ................
=>
```

图 30.4.4.8 下载的数据

在使用 winhex 软件来查看 zImage，检查一下前面的数据是否和图 30.4.4.8 只的一致，结果如图 30.4.4.9 所示：

zImage

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
00000000	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1	00	00	A0	E1
00000020	03	00	00	EA	18	28	6F	01	00	00	00	00	60	A3	5C	00	01	02	03	04	00	90	0F	E1	E8	04	00	EB	01	70	A0	E1
00000040	02	80	A0	E1	00	20	0F	E1	03	00	12	E3	01	00	00	1A	17	00	A0	E3	56	34	12	EF	00	00	0F	E1	1A	00	20	E2
00000060	1F	00	10	E3	1F	00	C0	E3	D3	00	80	E3	04	00	00	1A	01	0C	80	E3	0C	E0	8F	E2	00	F0	6F	E1	0E	F3	2E	E1
00000080	6E	00	60	E1	00	F0	21	E1	09	F0	6F	E1	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	0F	40	A0	E1	3E	43	04	E2	02	49	84	E2	0F	00	A0	E1	04	00	50	E1	AC	01	9F	35	0F	00	80	30	00	00	54	31
000000C0	01	40	84	33	6D	00	00	2B	5E	0F	8F	E2	4E	1C	90	E8	1C	D0	90	E5	01	00	40	E0	00	60	86	E0	00	A0	8A	E0
000000E0	00	90	DA	E5	01	E0	DA	E5	0E	94	89	E1	02	E0	DA	E5	03	A0	DA	E5	0E	98	89	E1	0A	9C	89	E1	00	D0	8D	E0
00000100	01	A8	8D	E2	00	50	A0	E3	01	A9	8A	E2	0A	00	54	E1	1E	00	00	2A	09	A0	84	E0	70	90	8F	E2	09	00	5A	E1

图 30.4.4.9 winhex 查看 zImage

可以看出图 30.4.4.8 和图 30.4.4.9 中的前 100 个字节的数据一致，说明 nfs 命令下载到的 zImage 是正确的。

4、tftp 命令

tftp 命令的作用和 nfs 命令一样，都是用于通过网络下载东西到 DRAM 中，只是 tftp 命令使用的 TFTP 协议，Ubuntu 主机作为 TFTP 服务器。因此需要在 Ubuntu 上搭建 TFTP 服务器，需要安装 tftp-hpa 和 tftpd-hpa，命令如下：

```
sudo apt-get install tftp-hpa tftpd-hpa
```

和 NFS 一样，TFTP 也需要一个文件夹来存放文件，在用户目录下新建一个目录，命令如下：

```
mkdir /home/zuozhongkai/linux/tftpboot
chmod 777 /home/zuozhongkai/linux/tftpboot
```

这样我就在我的电脑上创建了一个名为 tftpboot 的目录(文件夹)，路径为/home/zuozhongkai/linux/tftpboot。注意！我们要给 tftpboot 文件夹权限，否则的话 uboot 不能从 tftpboot 文件夹里面下载文件。

最后配置 tftp，打开文件安装完成以后新建文件/etc/xinetd.d/tftp，然后在里面输入如下内容：

示例代码 30.4.4.1 /etc/xinetd.d/tftp 文件内容

```
1  server tftp
2  {
3      socket_type    = dgram
4      protocol       = udp
5      wait           = yes
6      user           = root
7      server         = /usr/sbin/in.tftpd
8      server_args    = -s /home/zuozhongkai/linux/tftpboot/
9      disable        = no
10     per_source     = 11
11     cps            = 100 2
12     flags          = IPv4
13 }
```

完了以后启动 tftp 服务，命令如下：

```
sudo service tftpd-hpa start
```

打开/etc/default/tftpd-hpa 文件，将其修改为如下所示内容：

示例代码 30.4.4.2 /etc/default/tftpd-hpa 文件内容

```
1 # /etc/default/tftpd-hpa
2 
3 TFTP_USERNAME="tftp"
4 TFTP_DIRECTORY="/home/zuozhongkai/linux/tftpboot"
5 TFTP_ADDRESS=":69"
6 TFTP_OPTIONS="-l -c -s"
```

TFTP_DIRECTORY 就是我们上面创建的 tftp 文件夹目录，以后我们就将所有需要通过 TFTP 传输的文件都放到这个文件夹里面，并且要给予这些文件相应的权限。

最后输入如下命令， 重启 tftp 服务器：

```
sudo service tftpd-hpa restart
```

tftp 服务器已经搭建好了，接下来就是使用了。将 zImage 镜像文件拷贝到 tftpboot 文件夹中，并且给予 zImage 相应的权限，命令如下：

```
cp zImage /home/zuozhongkai/linux/tftpboot/
cd /home/zuozhongkai/linux/tftpboot/
chmod 777 zImage
```

万事俱备，只剩验证了，uboot 中的 tftp 命令格式如下：

```
tftpboot [loadAddress] [[hostIPaddr:]bootfilename]
```

看起来和 nfs 命令格式一样的，loadAddress 是文件在 DRAM 中的存放地址，[[hostIPaddr:]bootfilename]是要从 Ubuntu 中下载的文件。但是和 nfs 命令的区别在于，tftp 命令不需要输入文件在 Ubuntu 中的完整路径，只需要输入文件名即可。比如我们现在将 tftpboot 文件夹里面的 zImage 文件下载到开发板 DRAM 的 0X80800000 地址处，命令如下：

```
tftp 80800000 zImage
```

下载过程如图 30.4.4.10 所示：

```
=> tftp 80800000 zImage
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ########################
         1.4 MiB/s
done
Bytes transferred = 6071136 (5ca360 hex)
=>
```

图 30.4.4.10 tftp 命令下载过程

从图 30.4.4.10 可以看出，zImage 下载成功了，网速为 1.4MibB/s，文件大小为 6071136 字节。同样的，可以使用 md.b 命令来查看前 100 个字节的数据是否和图 30.4.4.9 中的相等。有时候使用 fttp 命令从 Ubuntu 中下载文件的时候会出现如图 30.4.4.11 所示的错误提示：

```
=> tftp 80800000 zImage
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'zImage'.
Load address: 0x80800000
Loading: *
TFTP error: 'Permission denied' (0)
Starting again
```

图 30.4.4.11 tftp 下载出错

在图 30.4.4.11 中可以看到“TFTP error: 'Permission denied' (0)”这样的错误提示，提示没有权限，出现这个错误一般有两个原因：

①、在 Ubuntu 中创建 tftpboot 目录的时候没有给予 tftboot 相应的权限。

②、tftpboot 目录中要下载的文件没有给予相应的权限。

针对上述两个问题，使用命令“chmod 777 xxx”来给予权限，其中“xxx”就是要给予权限的文件或文件夹。

好了，uboot 中关于网络的命令就讲解到这里，我们最常用的就是 ping、nfs 和 tftp 这三个命令。使用 ping 命令来查看网络的连接状态，使用 nfs 和 tftp 命令来从 Ubuntu 主机中下载文件。

30.4.5 EMMC 和 SD 卡操作命令

uboot 支持 EMMC 和 SD 卡，因此也要提供 EMMC 和 SD 卡的操作命令。一般认为 EMMC 和 SD 卡是同一个东西，所以没有特殊说明，本教程统一使用 MMC 来代指 EMMC 和 SD 卡。uboot 中常用于操作 MMC 设备的命令为“mmc”。

mmc 是一系列的命令，其后可以跟不同的参数，输入“？mmc”即可查看 mmc 有关的命令，如图 30.4.5.1 所示：

```
=> ? mmc
mmc - MMC sub system

Usage:
mmc info - display info of the current MMC device
mmc read addr blk# cnt
mmc write addr blk# cnt
mmc erase blk# cnt
mmc rescan
mmc part - lists available partition on current mmc device
mmc dev [dev] [part] - show or set current mmc device [partition]
mmc list - lists available devices
mmc hwpartition [args...] - does hardware partitioning
  arguments (sizes in 512-byte blocks):
    [user [enh start cnt] [wrrel {on|off}]] - sets user data area attributes
    [gp1|gp2|gp3|gp4 cnt [enh] [wrrel {on|off}]] - general purpose partition
    [check|set|complete] - mode, complete set partitioning completed
  WARNING: Partitioning is a write-once setting once it is set to complete.
  Power cycling is required to initialize partitions after set to complete.
mmc bootbus dev boot_bus_width reset_boot_bus_width boot_mode
 - Set the BOOT_BUS_WIDTH field of the specified device
mmc bootpart-resize <dev> <boot part size MB> <RPMB part size MB>
 - Change sizes of boot and RPMB partitions of specified device
mmc partconf dev boot_ack boot_partition partition_access
 - Change the bits of the PARTITION_CONFIG field of the specified device
mmc rst-function dev value
 - Change the RST_n_FUNCTION field of the specified device
   WARNING: This is a write-once field and 0 / 1 / 2 are the only valid values.
mmc setdsr <value> - set DSR register value

=>
```

图 30.4.5.1 mmc 命令

从图 30.4.5.1 可以看出，mmc 后面跟不同的参数可以实现不同的功能，如表 30.4.5.1 所示：

命令	描述
mmc info	输出 MMC 设备信息
mmc read	读取 MMC 中的数据。
mmc wirte	向 MMC 设备写入数据。
mmc rescan	扫描 MMC 设备。
mmc part	列出 MMC 设备的分区。
mmc dev	切换 MMC 设备。
mmc list	列出当前有效的所有 MMC 设备。
mmc hwpartition	设置 MMC 设备的分区。
mmc bootbus……	设置指定 MMC 设备的 BOOT_BUS_WIDTH 域的值。
mmc bootpart……	设置指定 MMC 设备的 boot 和 RPMB 分区的大小。
mmc partconf……	设置指定 MMC 设备的 PARTITION_CONFG 域的值。
mmc rst	复位 MMC 设备
mmc setdsr	设置 DSR 寄存器的值。

表 30.4.5.1 mmc 命令

1、mmc info 命令

mmc info 命令用于输出当前选中的 mmc info 设备的信息，输入命令“mmc info”即可，如图 30.4.5.2 所示：

```
=> mmc info
Device: FSL_SDHC
Manufacturer ID: 45
OEM: 100
Name: SEM04
Tran Speed: 52000000
Rd Block Len: 512
MMC version 4.5
High Capacity: Yes
Capacity: 3.7 GiB
Bus Width: 8-bit
Erase Group Size: 256 KiB
HC WP Group Size: 8 MiB
User Capacity: 3.7 GiB WRREL
Boot Capacity: 2 MiB
RPMB Capacity: 2 MiB
=>
```

图 30.4.5.2 mmc info 命令

从图 30.4.5.2 可以看出，当前选中的 MMC 设备是 EMMC，版本为 4.5，容量为 3.7GiB(EMMC 为 4GB)，速度为 52000000Hz=52MHz，8 位宽的总线。还有一个与 mmc info 命令相同功能的命令：mmcinfo，“mmc”和“info”之间没有空格。

2、mmc rescan 命令

mmc rescan 命令用于扫描当前开发板上所有的 MMC 设备，包括 EMMC 和 SD 卡，输入“mmc rescan”即可。

3、mmc list 命令

mmc list 命令用于来查看当前开发板一共有几个 MMC 设备，输入“mmc list”，结果如图 30.4.5.3 所示：

```
=> mmc list
FSL_SDHC: 0
FSL_SDHC: 1 (eMMC)
=>
```

图 30.4.5.3 扫描 MMC 设备

可以看出当前开发板有两个 MMC 设备：FSL_SDHC:0 和 FSL_SDHC:1 (eMMC)，这是因为我现在用的是 EMMC 版本的核心板，加上 SD 卡一共有两个 MMC 设备，FSL_SDHC:0 是 SD 卡，FSL_SDHC:1(eMMC)是 EMMC,。默认会将 EMMC 设置为当前 MMC 设备，这就是为什么输入“mmc info”查询到的是 EMMC 设备信息，而不是 SD 卡。要想查看 SD 卡信息，就要使用命令“mmc dev”来将 SD 卡设置为当前的 MMC 设备。

4、mmc dev 命令

mmc dev 命令用于切换当前 MMC 设备，命令格式如下：

```
mmc dev [dev] [part]
```

[dev]用来设置要切换的 MMC 设备号，[part]是分区号。如果不写分区号的话默认为分区 0。使用如下命令切换到 SD 卡：

```
mmc dev 0          //切换到 SD 卡，0 为 SD 卡，1 为 eMMC
```

结果如图 30.4.5.4 所示：

```
=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
```

图 30.4.5.4 切换到 SD 卡

从图 30.4.5.4 可以看出，切换到 SD 卡成功，mmc0 为当前的 MMC 设备，输入命令“mmc info”即可查看 SD 卡的信息，结果如图 30.4.5.5 所示：

```
=> mmc info
Device: FSL_SDHC
Manufacturer ID: 3
OEM: 5344
Name: SC16G
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
=>
```

图 30.4.5.5 SD 信息

从图 30.4.5.5 可以看出当前 SD 卡为 3.0 版本的，容量为 14.8GiB(16GB 的 SD 卡)，4 位宽的总线。

5、mmc part 命令

有时候 SD 卡或者 EMMC 会有多个分区，可以使用命令“mmc part”来查看其分区，比如查看 EMMC 的分区情况，输入如下命令：

```
mmc dev 1      //切换到 EMMC
mmc part       //查看 EMMC 分区
```

结果如图 30.4.5.6 所示：

```
=> mmc dev 1
switch to partitions #0, OK
mmc1(part 0) is current device
=> mmc part

Partition Map for MMC device 1  --   Partition Type: DOS

Part    Start Sector    Num Sectors     UUID            Type
  1     20480           1024000         488db264-01     0c
  2     1228800         6504448         488db264-02     83
=>
```

图 30.4.5.6 查看 EMMC 分区

从图 30.4.5.6 中可以看出，此时 EMMC 有两个分区，扇区 20480~1024000 为第一个分区，扇区 1228800~6504448 为第二个分区。如果 EMMC 里面烧写了 Linux 系统的话，EMMC 是有 3 个分区的，第 0 个分区存放 uboot，第 1 个分区存放 Linux 镜像文件和设备树，第 2 个分区存放根文件系统。但是在图 30.4.5.6 中只有两个分区，那是因为第 0 个分区没有格式化，所以识别不出来，实际上第 0 个分区是存在的。一个新的 SD 卡默认只有一个分区，那就是分区 0，所以前面讲解的 uboot 烧写到 SD 卡，其实就是将 u-boot.bin 烧写到了 SD 卡的分区 0 里面。后面学习 Linux 内核移植的时候再讲解怎么在 SD 卡中创建并格式化第二个分区，并将 Linux 镜像文件和设备树文件存放到第二个分区中。

如果要将 EMMC 的分区 2 设置为当前 MMC 设置，可以使用如下命令：

```
mmc dev 1 2
```

结果如图 30.4.5.7 所示：

```
=> mmc dev 1 2
switch to partitions #2, OK
mmc1(part 2) is current device
=>
```

图 30.4.5.7 设置 EMMC 分区 2 为当前设备

6、mmc read 命令

mmc read 命令用于读取 mmc 设备的数据，命令格式如下：

```
mmc read addr blk# cnt
```

addr 是数据读取到 DRAM 中的地址，blk 是要读取的块起始地址(十六进制)，一个块是 512 字节，这里的块和扇区是一个意思，在 MMC 设备中我们通常说扇区，cnt 是要读取的块数量(十六进制)。比如从 EMMC 的第 1536(0x600)个块开始，读取 16(0x10)个块的数据到 DRAM 的 0X80800000 地址处，命令如下：

```
mmc dev 1 0                    //切换到 MMC 分区 0
mmc read 80800000 600 10       //读取数据
```

结果如图 30.4.5.8 所示：

```
=> mmc dev 1 0
switch to partitions #0, OK
mmc1(part 0) is current device
=> mmc read 80800000 600 10

MMC read: dev # 1, block # 1536, count 16 ... 16 blocks read: OK
=>
```

图 30.4.5.8 mmc read 命令

这里我们还看不出来读取是否正确，通过 md.b 命令查看 0x80800000 处的数据就行了，查看 16*512=8192(0x2000)个字节的数据，命令如下：

```
md.b 80800000 2000
```

结果如图 30.4.5.9 所示：

```
=> mmc read 80800000 600 10

MMC read: dev # 1, block # 1536, count 16 ... 16 blocks read: OK
=> md.b 80800000 2000
80800000: 44 24 ec 88 62 61 75 64 72 61 74 65 3d 31 31 35    D$..baudrate=115
80800010: 32 30 30 00 62 6f 61 72 64 5f 6e 61 6d 65 3d 45    200.board_name=E
80800020: 56 4b 00 62 6f 61 72 64 5f 72 65 76 3d 31 34 58    VK.board_rev=14X
80800030: 31 34 00 62 6f 6f 74 5f 66 64 74 3d 74 72 79 00    14.boot_fdt=try.
80800040: 62 6f 6f 74 61 72 67 73 3d 63 6f 6e 73 6f 6c 65    bootargs=console
80800050: 3d 74 74 79 6d 78 63 30 2c 31 31 35 32 30 30 20    =ttymxc0,115200 
80800060: 72 6f 6f 74 3d 2f 64 65 76 2f 6e 66 73 20 72 77    root=/dev/nfs rw
80800070: 20 6e 66 73 3d 2f 64 65 76 2f 6e 66 73 20 72 77     nfs=/dev/nfs rw
80800080: 20 6e 66 73 72 6f 6f 74 3d 31 39 32 2e 31 36 38     nfsroot=192.168
80800090: 2e 31 2e 32 35 30 3a 2f 68 6f 6d 65 2f 7a 75 6f    .1.250:/home/zuo
808000a0: 7a 68 6f 6e 67 6b 61 69 2f 6c 69 6e 75 78 2f 6e    zhongkai/linux/n
808000b0: 66 73 2f 69 6d 78 5f 72 6f 6f 74 66 73 20 69 70    fs/imx_rootfs ip
808000c0: 3d 31 39 32 2e 31 36 38 2e 31 2e 35 35 3a 31 39    =192.168.1.55:19
808000d0: 32 2e 31 36 38 2e 31 2e 32 35 30 3a 31 39 32 2e    2.168.1.250:192.
808000e0: 31 36 38 2e 31 2e 31 3a 32 35 35 2e 32 35 35 2e    168.1.1:255.255.
808000f0: 32 35 35 2e 30 3a 3a 65 74 68 30 3a 6f 66 66 00    255.0::eth0:off.
80800100: 62 6f 6f 74 63 6d 64 3d 6e 66 73 20 38 30 38 30    bootcmd=nfs 8080
80800110: 30 30 30 30 20 31 39 32 2e 31 36 38 2e 31 2e 32    0000 192.168.1.2
80800120: 35 30 3a 2f 68 6f 6d 65 2f 7a 75 6f 7a 68 6f 6e    50:/home/zuozhon
80800130: 67 6b 61 69 2f 6c 69 6e 75 78 2f 6e 66 73 2f 7a    gkai/linux/nfs/z
80800140: 49 6d 61 67 65 5f 69 6d 78 36 75 6c 3b 6e 66 73    Image_imx6ul;nfs
80800150: 20 38 33 30 30 30 30 30 30 20 31 39 32 2e 31 36     83000000 192.16
```

图 30.4.5.9 读取到的数据(部分截图)

从图 30.4.5.9 可以看到“D$..baudrate=115200.board_name=EVK.board_rev=14X14.”等字样，这个就是 uboot 中的环境变量。EMMC 核心板 uboot 环境变量的存储起始地址就是 1536*512=786432。

7、mmc write 命令

要将数据写到 MMC 设备里面，可以使用命令“mmc write”，格式如下：

```
mmc write addr blk# cnt
```

addr 是要写入 MMC 中的数据在 DRAM 中的起始地址，blk 是要写入 MMC 的块起始地址(十六进制)，cnt 是要写入的块大小，一个块为 512 字节。我们可以使用命令“mmc write”来升级 uboot，也就是在 uboot 中更新 uboot。这里要用到 nfs 或者 tftp 命令，通过 nfs 或者 tftp 命令将新的 u-boot.bin 下载到开发板的 DRAM 中，然后再使用命令“mmc write”将其写入到 MMC 设备中。我们就来更新一下 SD 中的 uboot，先查看一下 SD 卡中的 uboot 版本号，注意编译时间，输入命令：

```
mmc dev 0      //切换到 SD 卡
version        //查看版本号
```

结果如图 30.4.5.10 所示：

```
=> mmc dev 1
switch to partitions #0, OK
mmc1(part 0) is current device
=> version

U-Boot 2016.03 (Apr 15 2019 - 12:52:04 +0800)
arm-linux-gnueabihf-gcc (Linaro GCC 4.9-2017.01) 4.9.4
GNU ld (Linaro_Binutils-2017.01) 2.24.0.20141017 Linaro 2014_11-3-git
=>
```

图 30.4.5.10 uboot 版本查询

可以看出当前 SD 卡中的 uboot 是 2019 年 4 月 15 日 12:52:04 编译的。我们现在重新编译一下 uboot，然后将编译出来的 u-boot.imx(u-boot.bin 前面加了一些头文件)拷贝到 Ubuntu 中的 tftpboot 目录下。最后使用 tftp 命令将其下载到 0x80800000 地址处，命令如下：

```
tftp 80800000 u-boot.imx
```

下载过程如图 30.4.5.11 所示：

```
=> tftp 80800000 u-boot.imx
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'u-boot.imx'.
Load address: 0x80800000
Loading: #############################
         2.2 MiB/s
done
Bytes transferred = 416768 (65c00 hex)
=>
```

图 30.4.5.11 u-boot.imx 下载过程

可以看出，u-boot.imx 大小为 416768 字节，416768/512=814，所以我们要向 SD 卡中写入 814 个块，如果有小数的话就要加 1 个块。使用命令“mmc write”从 SD 卡分区 0 第 2 个块(扇区)开始烧写，一共烧写 814(0x32E)个块，命令如下：

```
mmc dev 0 0
mmc write 80800000 2 32E
```

烧写过程如图 30.4.5.12 所示：

```
=> mmc dev 0 0
switch to partitions #0, OK
mmc0 is current device
=> mmc write 80800000 2 32E

MMC write: dev # 0, block # 2, count 814 ... 814 blocks written: OK
=>
```

图 30.4.5.12 烧写过程

烧写成功，重启开发板(从 SD 卡启动)，重启以后再输入 version 来查看版本号，结果如图 30.4.5.13 所示：

```
=> version

U-Boot 2016.03 (Apr 21 2019 - 18:05:59 +0800)
arm-linux-gnueabihf-gcc (Linaro GCC 4.9-2017.01) 4.9.4
GNU ld (Linaro_Binutils-2017.01) 2.24.0.20141017 Linaro 2014_11-3-git
=>
```

图 30.4.5.13 uboot 版本号

从图 30.4.5.13 可以看出，此时的 uboot 是 2019 年 4 月 21 号 18:05:59 编译的，这个时间就是我刚刚编译 uboot 的时间，说明 uboot 更新成功。这里我们就学会了如何在 uboot 中更新 uboot 了，如果要更新 EMMC 中的 uboot 也是一样的。

千万不要写 SD 卡或者 EMMC 的前两个块(扇区)，里面保存着分区表！

千万不要写 SD 卡或者 EMMC 的前两个块(扇区)，里面保存着分区表！

千万不要写 SD 卡或者 EMMC 的前两个块(扇区)，里面保存着分区表！

8、mmc erase 命令

如果要擦除 MMC 设备的指定块就是用命令“mmc erase”，命令格式如下：

```
mmc erase blk# cnt
```

blk 为要擦除的起始块，cnt 是要擦除的数量。没事不要用 mmc erase 来擦除 MMC 设备！！！

关于 MMC 设备相关的命令就讲解到这里，表 30.4.5.1 中还有一些跟 MMC 设备操作有关的命令，但是很少用到，这里就不讲解了，感兴趣的可以上网查一下，或者在 uboot 中查看这些命令的使用方法。

30.4.6 FAT 格式文件系统操作命令

有时候需要在 uboot 中对 SD 卡或者 EMMC 中存储的文件进行操作，这时候就要用到文件操作命令，跟文件操作相关的命令有：fatinfo、fatls、fstype、fatload 和 fatwrite，但是这些文件操作命令只支持 FAT 格式的文件系统！！

1、fatinfo 命令

fatinfo 命令用于查询指定 MMC 设置指定分区的文件系统信息，格式如下：

```
fatinfo <interface> [<dev[:part]>]
```

interface 表示接口，比如 mmc，dev 是查询的设备号，part 是要查询的分区。比如我们要查询 EMMC 分区 1 的文件系统信息，命令如下：

```
fatinfo mmc 1:1
```

结果如图 30.4.6.1 所示：

```
=> fatinfo mmc 1:1
Interface:  MMC
  Device 1: Vendor: Man 000045 Snr 91ef8153 Rev: 3.10 Prod: SEM04G
            Type: Removable Hard Disk
            Capacity: 3776.0 MB = 3.6 GB (7733248 x 512)
Filesystem: FAT16 "NO NAME    "
=>
```

图 30.4.6.1 emmc 分区 1 文件系统信息

从上图可以看出，EMMC 分区 1 的文件系统为 FAT16 格式的。

2、fatls 命令

fatls 命令用于查询 FAT 格式设备的目录和文件信息，命令格式如下：

```
fatls <interface> [<dev[:part]>] [directory]
```

interface 是要查询的接口，比如 mmc，dev 是要查询的设备号，part 是要查询的分区，directory 是要查询的目录。比如查询 EMMC 分区 1 中的所有的目录和文件，输入命令：

```
fatls mmc 1:1
```

结果如图 30.4.6.2 所示：

```
=> fatls mmc 1:1
  6071136   zimage
    37099   imx6ull-14x14-evk.dtb

2 file(s), 0 dir(s)

=>
```

图 30.4.6.2 EMMC 分区 1 文件查询

从上图可以看出，emmc 的分区 1 中存放着两个文件：zimage 和 imx6ull-14x14-evk.dtb，这两个文件分别是 linux 镜像文件和设备树。并且在 emmc 的分区 1 中有两个文件，没有目录

3、fstype 命令

fstype 用于查看 MMC 设备某个分区的文件系统格式，命令格式如下：

```
fstype <interface> <dev>:<part>
```

正点原子 EMMC 核心板上的 EMMC 默认有 3 个分区，我们来查看一下这三个分区的文件系统格式，输入命令：

```
fstype mmc 1:0
fstype mmc 1:1
fstype mmc 1:2
```

结果如图 30.4.6.3 所示：

```
=> fstype mmc 1:0
Failed to mount ext2 filesystem...
** Unrecognized filesystem type **
=> fstype mmc 1:1
fat
=> fstype mmc 1:2
ext4
=>
```

图 30.4.6.3 fstype 命令

从上图可以看出，分区 0 格式未知，因为分区 0 存放的 uboot，并且分区 0 没有格式化，所以文件系统格式未知。分区 1 的格式为 fat，分区 1 用于存放 linux 镜像和设备树。分区 2 的格式为 ext4，用于存放 Linux 的跟文件系统。

4、fatload 命令

fatload 命令用于将指定的文件读取到 DRAM 中，命令格式如下：

```
fatload <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]]]]]
```

interface 为接口，比如 mmc，dev 是设备号，part 是分区，addr 是保存在 DRAM 中的起始地址，filename 是要读取的文件名字。bytes 表示读取多少字节的数据，如果 bytes 为 0 或者省略的话表示读取整个文件。pos 是要读的文件相对于文件首地址的偏移，如果为 0 或者省略的话表示从文件首地址开始读取。我们将 EMMC 分区 1 中的 zImage 文件读取到 DRAM 中的 0X80800000 地址处，命令如下：

```
fatload mmc 1:1 80800000 zImage
```

操作过程如图 30.4.6.4 所示：

```
=> fatload mmc 1:1 80800000 zImage
reading zImage
6071136 bytes read in 153 ms (37.8 MiB/s)
=>
```

图 30.4.6.4 读取过程

从上图可以看出在 153ms 内读取了 6071136 个字节的数据，速度为 37.8MiB/s，速度是非常快的，因为这是从 EMMC 里面读取的，而 EMMC 是 8 位的，速度肯定会很快的。

5、fatwrite 命令

fatwirte 命令用于将 DRAM 中的数据写入到 MMC 设备中，命令格式如下：

```
fatwrite <interface> <dev[:part]> <addr> <filename> <bytes>
```

interface 为接口，比如 mmc，dev 是设备号，part 是分区，addr 是要写入的数据在 DRAM 中的起始地址，filename 是写入的数据文件名字，bytes 表示要写入多少自己的数据。我们可以通过 fatwrite 命令在 uboot 中更新 linux 镜像文件和设备树。我们以更新 linux 镜像文件 zImage 为例，首先将正点原子 I.MX6U-ALPHA 开发板提供的 zImage 镜像文件拷贝到 Ubuntu 中的 tftpboot 目录下，zImage 镜像文件放到了开发板光盘中，路径为：开发板光盘->8、开发板系统镜像->zImage。拷贝完成以后的 tftpboot 目录如图 30.4.6.5 所示：

```
zuozhongkai@ubuntu:~/linux/tftpboot$ ls
u-boot.imx  zImage
zuozhongkai@ubuntu:~/linux/tftpboot$
```

图 30.4.6.5 zImage 放到 tftpboot 目录中。

使用命令 tftp 将 zImage 下载到 DRAM 的 0X80800000 地址处，命令如下：

```
tftp 80800000 zImage
```

下载过程如图 30.4.6.6 所示：

```
=> tftp 80800000 zImage
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ########################
         2.2 MiB/s
done
Bytes transferred = 6039328 (5c2720 hex)
=>
```

图 30.4.6.6 zImage 下载过程

zImage 大小为 6039328(0X5C2720)个字节，接下来使用命令 fatwrite 将其写入到 EMMC 的分区 1 中，文件名字为 zImage，命令如下：

```
fatwrite mmc 1:1 80800000 zImage 0x5c2720
```

结果如图 30.4.6.7 所示：

```
=> fatwrite mmc 1:1 80800000 zImage 0x5c2720
writing zImage
6039328 bytes written
=>
```

图 30.4.6.7 将 zImage 烧写到 EMMC 扇区 1 中

完成以后使用“fatls”命令查看一下 EMMC 分区 1 里面的文件，结果如图 30.4.6.8 所示：

```
=> fatls mmc 1:1
  6039328   zimage
    37099   imx6ull-14x14-evk.dtb

2 file(s), 0 dir(s)

=>
```

图 30.4.6.8 EMMC 分区 1 里面的文件

30.4.7 EXT 格式文件系统操作命令

uboot 有 ext2 和 ext4 这两种格式的文件系统的操作命令，常用的就四个命令，分别为：ext2load、ext2ls、ext4load、ext4ls 和 ext4write。这些命令的含义和使用与 fatload、fatls 和 fatwrit 一样，只是 ext2 和 ext4 都是针对 ext 文件系统的。比如 ext4ls 命令，EMMC 的分区 2 就是 ext4 格式的，使用 ext4ls 就可以查询 EMMC 的分区 2 中的文件和目录，输入命令：

```
ext4ls mmc 1:2
```

结果如图 30.4.7.1 所示：

```
=> ext4ls mmc 1:2
<DIR>       4096 .
<DIR>       4096 ..
<DIR>      16384 lost+found
<DIR>       4096 photos
            6147 .ash_history
<DIR>       4096 sbin
<DIR>       4096 tmp
<DIR>       4096 unit_tests
<DIR>       4096 lib
              22 test.txt
<DIR>       4096 opt
<DIR>       4096 mnt
<DIR>       4096 dev
<DIR>       4096 usr
<DIR>       4096 etc
<DIR>       4096 .mplayer
<DIR>       4096 .cache
<DIR>       4096 picture
<DIR>       4096 root
<DIR>       4096 .config
<DIR>       4096 qt_demo
<DIR>       4096 media
<DIR>       4096 drivers
<DIR>       4096 proc
<DIR>       4096 sys
<DIR>       4096 var
<DIR>       4096 share
<SYM>         11 linuxrc
<DIR>       4096 bin
<DIR>       4096 music
<DIR>       4096 .local
<DIR>       4096 钱钟书文集
<DIR>       4096 include
<DIR>       4096 mjpg-streamer
=>
```

图 30.4.7.1 ext4ls 命令

关于 ext 格式文件系统其他命令的操作参考 30.4.6 小节的即可，这里就不讲解了。

30.4.8 NAND 操作命令

uboot 是支持 NAND Flash 的，所以也有 NAND Flash 的操作命令，前提是使用的 NAND 版本的核心板，并且编译 NAND 核心板对应的 uboot，然后使用 imxdownload 软件将 u-boot.bin 烧写到 SD 卡中，最后通过 SD 卡启动。一般情况下 NAND 版本的核心板已经烧写好了 uboot、linux kernel 和 rootfs 这些文件，所以可以将 BOOT 拨到 NAND，然后直接从 NAND Flash 启动即可。

NAND 版核心板启动信息如图 30.4.8.1 所示：

```
U-Boot 2016.03 (May 24 2019 - 10:31:09 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 52C
Reset cause: POR
Board: MX6ULL ALIENTEK NAND
I2C:   ready
DRAM:  512 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
*** Warning - bad CRC, using default environment

Display: TFT7016 (1024x600)
Video: 1024x600x24
In:    serial
Out:   serial
Err:   serial
Net:   FEC1
Error: FEC1 address not set.

Normal Boot
Hit any key to stop autoboot:  0
=>
```

图 30.4.8.1 NAND 核心板启动信息

从图 30.4.8.1 可以看出，当前开发板的 NAND 容量为 512MiB。输入“? nand”即可查看所有有关 NAND 令，如图 30.4.8.2 所示：

```
=> ? nand
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
    read/write 'size' bytes starting at offset 'off'
    to/from memory address 'addr', skipping bad blocks.
nand read.raw - addr off|partition [count]
nand write.raw - addr off|partition [count]
    Use read.raw/write.raw to avoid ECC and access the flash as-is.
nand write.trimffs - addr off|partition size
    write 'size' bytes starting at offset 'off' from memory address
    'addr', skipping bad blocks and dropping any pages at the end
    of eraseblocks that contain only 0xFF
nand erase[.spread] [clean] off size - erase 'size' bytes from offset 'off'
    With '.spread', erase enough for given file size, otherwise,
    'size' includes skipped bad blocks.
nand erase.part [clean] partition - erase entire mtd partition'
nand erase.chip [clean] - erase entire chip'
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub [-y] off size | scrub.part partition | scrub.chip
    really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
=>
```

图 30.4.8.2 NAND 相关操作命令

可以看出，NAND 相关的操作命令少，本节我们讲解一些常用的命令。

1、nand info 命令

此命令用户打印 NAND Flash 信息，输入“nand info”，结果如图 30.4.8.3 所示：

```
=> nand info

Device 0: nand0, sector size 128 KiB
  Page size        2048 b
  OOB size          128 b
  Erase size     131072 b
  subpagesize      2048 b
  options    0x40000200
  bbt options 0x      8000
=>
```

图 30.4.8.3 nand 信息

图 30.4.8.3 中给出了 NAND 的页大小、OOB 域大小，擦除大小等信息。可以对照着所使用的 NAND Flash 数据手册来查看一下这些信息是否正确。

2、nand device 命令

nand device 用于切换 NAND Flash，如果你的板子支持多片 NAND 的话就可以使用此命令来设置当前所使用的 NAND。这个需要你的 CPU 有两个 NAND 控制器，并且两个 NAND 控制器各接一片 NAND Flash。就跟 I.MX6U 有两个 SDIO 接口，这两个 SDIO 接口可以接两个 MMC 设备一样。不过一般情况下 CPU 只有一个 NAND 接口，而且在使用中只接一片 NAND。

3、nand erase 命令

nand erase 命令用于擦除 NAND Flash，NAND Flash 的特性决定了在向 NAND Flash 写数据之前一定要先对要写入的区域进行擦除。“nand erase”命令有三种形式：

```
nand erase[.spread] [clean] off size //从指定地址开始(off)开始，擦除指定大小(size)的区域。
nand erase.part [clean] partition    //擦除指定的分区
nand erase.chip [clean]              //全篇擦除
```

NAND 的擦除命令一般是配合写命令的，后面讲解 NAND 写命令的时候在演示如何使用“nand erase”。

4、nand write 命令

此命令用于向 NAND 指定地址写入指定的数据，一般和“nand erase”命令配置使用来更新 NAND 中的 uboot、linux kernel 或设备树等文件，命令格式如下：

```
nand write addr off size
```

addr 是要写入的数据首地址，off 是 NAND 中的目的地址，size 是要写入的数据大小。

以更新 NAND 中的 uboot 为例，讲解一下如何使用此命令。先编译出来 NAND 版本的 u-boot.imx 文件，在烧写之前要先对 NAND 进行分区，也就是规划好 uboot、linux kernel、设备树和根文件系统的存储区域，I.MX6U-ALPHA 开发板的 NAND 分区如下：

```
0x000000000000-0x000004000000 : "boot"
0x000004000000-0x000006000000 : "kernel"
0x000006000000-0x000007000000 : "dtb"
0x000007000000-0x000020000000 : "rootfs"
```

一共有四个分区，第一个分区存放 uboot，地址范围为 0x0~0x4000000(共 64MB)；第二个分区存放 kernel(也就是 linux kernel)，地址范围为 0x4000000~0x6000000(共 32MB)；第三个分区存放 dtb(设备树)，地址范围为 0x6000000~0x7000000(共 16MB)；剩下的所有存储空间全部作为最后一个分区，存放 rootfs(根文件系统)。

uboot 是从地址 0 开始存放的，其实用不了这么大的区域，但是为了好管理才分配了这么大的，将 NAND 版本的 u-boot.imx 文件放到 Ubuntu 中的 tftpboot 目录中，然后使用 tftp 命令将其下载到开发板的 0X87800000 地址处，最终使用“nand write”将其烧写到 NAND 中，命令如下：

```
tftp 0x87800000 u-boot.imx              //下载 u-boot.imx 到 DRAM 中
```

```
nand erase 0x0 0x100000                   //从地址 0 开始擦除 1MB 的空间
nand write 0x87800000 0x0 0x100000  //将接收到的 u-boot.imx 写到 NAND 中
```

u-boot.imx 很小，一般就是 4,5 百 KB，所以擦除 1MB 的空间就可以了。写入的时候也是按照 1M 的数据写入的，所以肯定会写入一些无效的数据。你也可以将写入的大小改为 u-boot.imx 这个文件的大小，这样写入的数据量就是 u-boot.imx 的实际大小了。

同理我们也可以更新 NAND 中的 linux kernel 和设备树(dtb)文件，命令如下：

```
tftp 0x87800000 zImage                          //下载 zImage 到 DRAM 中
nand erase 0x4000000 0xA00000                   //从地址 0x4000000 开始擦除 10MB 的空间
nand write 0x87800000 0x4000000 0xA00000        //将接收到的 zImage 写到 NAND 中
```

这里我们擦出了 10MB 的空间，因为一般 zImage 就是 6,7MB 左右，10MB 肯定够了，如果不够的话就将在多擦除一点就行了。

最后烧写设备树(dtb)文件文件，命令如下：

```
tftp 0x87800000 imx6ull-alientek-nand.dtb       //下载 dtb 到 DRAM 中
nand erase 0x6000000 0x100000                   //从地址 0x6000000 开始擦除 1MB 的空间
nand write 0x87800000 0x6000000 0x100000        //将接收到的 dtb 写到 NAND 中
```

dtb 文件一般只有几十 KB，所以擦除 1M 是绰绰有余的了。

根文件系统(rootfs)就不要在 uboot 中更新了，还是使用 NXP 提供的 MFGTool 工具来烧写，因为根文件系统太大！很有可能超过开发板 DRAM 的大小，这样连下载都没法下载，更别说更新了。

4、nand read 命令

此命令用于从 NAND 中的指定地址读取指定大小的数据到 DRAM 中，命令格式如下：

```
nand read addr off size
```

addr 是目的地址，off 是要读取的 NAND 中的数据源地址，size 是要读取的数据大小。比如我们读取设备树(dtb)文件到 0x83000000 地址处，命令如下：

```
nand read 0x83000000 0x6000000 0x19000
```

过程如图 30.4.8.4 所示：

```
=> nand read 0x83000000 0x6000000 0x19000

NAND read: device 0 offset 0x6000000, size 0x19000
 102400 bytes read: OK
=>
```

图 30.4.8.4 nand read 读取过程

设备树文件读取到 DRAM 中以后就可以使用 fdt 命令来对设备树进行操作了，首先设置 fdt 的地址，fdt 地址就是 DRAM 中设备树的首地址，命令如下：

```
fdt addr 83000000
```

设置好以后可以使用“fdt header”来查看设备树的头信息，输入命令：

```
fdt header
```

结果如图 30.4.8.5 所示：

```
=> fdt header
magic:                  0xd00dfeed
totalsize:              0x9435 (37941)
off_dt_struct:          0x38
off_dt_strings:         0x89f4
off_mem_rsvmap:         0x28
version:                17
last_comp_version:      16
boot_cpuid_phys:        0x0
size_dt_strings:        0xa41
size_dt_struct:         0x89bc
number mem_rsv:         0x0

=>
```

图 30.4.8.5 设备树头信息

输入命令“fdt print”就可以查看设备树文件的内容，输入命令：

```
fdt print
```

结果如图 30.4.8.6 所示：

```
=> fdt print
/ {
        #address-cells = <0x00000001>;
        #size-cells = <0x00000001>;
        model = "Freescale i.MX6 UltraLite 14x14 EVK Board";
        compatible = "fsl,imx6ul-14x14-evk", "fsl,imx6ul";
        chosen {
                stdout-path = "/soc/aips-bus@02000000/spba-bus@02000000/serial@02020000";
        };
        aliases {
                can0 = "/soc/aips-bus@02000000/can@02090000";
                can1 = "/soc/aips-bus@02000000/can@02094000";
                ethernet0 = "/soc/aips-bus@02100000/ethernet@02188000";
                ethernet1 = "/soc/aips-bus@02000000/ethernet@020b4000";
                gpio0 = "/soc/aips-bus@02000000/gpio@0209c000";
                gpio1 = "/soc/aips-bus@02000000/gpio@020a0000";
                gpio2 = "/soc/aips-bus@02000000/gpio@020a4000";
                gpio3 = "/soc/aips-bus@02000000/gpio@020a8000";
                gpio4 = "/soc/aips-bus@02000000/gpio@020ac000";
                i2c0 = "/soc/aips-bus@02100000/i2c@021a0000";
                i2c1 = "/soc/aips-bus@02100000/i2c@021a4000";
                i2c2 = "/soc/aips-bus@02100000/i2c@021a8000";
                i2c3 = "/soc/aips-bus@02100000/i2c@021f8000";
                mmc0 = "/soc/aips-bus@02100000/usdhc@02190000";
                mmc1 = "/soc/aips-bus@02100000/usdhc@02194000";
                serial0 = "/soc/aips-bus@02000000/spba-bus@02000000/serial@02020000";
                serial1 = "/soc/aips-bus@02100000/serial@021e8000";
                serial2 = "/soc/aips-bus@02100000/serial@021ec000";
                serial3 = "/soc/aips-bus@02100000/serial@021f0000";
                serial4 = "/soc/aips-bus@02100000/serial@021f4000";
                serial5 = "/soc/aips-bus@02100000/serial@021fc000";
                serial6 = "/soc/aips-bus@02000000/spba-bus@02000000/serial@02018000";
                serial7 = "/soc/aips-bus@02000000/spba-bus@02000000/serial@02024000";
                spi0 = "/soc/aips-bus@02000000/spba-bus@02000000/ecspi@02008000";
                spi1 = "/soc/aips-bus@02000000/spba-bus@02000000/ecspi@0200c000";
```

图 30.4.8.6 设备树文件

图 30.4.8.6 中的文件就是我们写到 NAND 中的设备树文件，至于设备树文件的详细内容我们后面会有专门的章节来讲解，这里大家知道这个文件就行了。

NAND 常用的操作命令就是擦除、读和写，至于其他的命令大家可以自行研究一下，一定不要尝试全片擦除 NAND 的指令!! 否则 NAND 就被全部擦除掉了，什么都没有了，又得重头烧整个系统。

30.4.9 BOOT 操作命令

uboot 的本质工作是引导 Linux，所以 uboot 肯定有相关的 boot(引导)命令来启动 Linux。常用的跟 boot 有关的命令有：bootz、bootm 和 boot。

1、bootz 命令

要启动 Linux，需要先将 Linux 镜像文件拷贝到 DRAM 中，如果使用到设备树的话也需要将设备树拷贝到 DRAM 中。可以从 EMMC 或者 NAND 等存储设备中将 Linux 镜像和设备树文件拷贝到 DRAM，也可以通过 nfs 或者 tftp 将 Linux 镜像文件和设备树文件下载到 DRAM 中。

不管用那种方法，只要能将 Linux 镜像和设备树文件存到 DRAM 中就行，然后使用 bootz 命令来启动，bootz 命令用于自动 zImage 镜像文件，bootz 命令格式如下：

```
bootz [addr [initrd[:size]] [fdt]]
```

命令 bootz 有三个参数，addr 是 Linux 镜像文件在 DRAM 中的位置，initrd 是 initrd 文件在 DRAM 中的地址，如果不使用 initrd 的话使用‘-’代替即可，fdt 就是设备树文件在 DRAM 中的地址。现在我们使用网络和 EMMC 两种方法来启动 Linux 系统，首先将 I.MX6U-ALPHA 开发板的 Linux 镜像和设备树发送到 Ubuntu 主机中的 tftpboot 文件夹下。Linux 镜像文件前面已经放到了 tftpboot 文件夹中，现在把设备树文件放到 tftpboot 文件夹里面。以 EMMC 核心板为例，将开发板光盘->8、开发板系统镜像->imx6ull-alientek-emmc.dtb 文件发送到 Ubuntu 主机中的 tftpboot 文件夹里面，完成以后的 tftpboot 文件夹如图 30.4.9.1 所示：

```
zuozhongkai@ubuntu:~/linux/tftpboot$ ls
imx6ull-alientek-emmc.dtb  u-boot.imx  zImage
zuozhongkai@ubuntu:~/linux/tftpboot$
```

图 30.4.9.1 tftpboot 文件夹

下载 Linux 镜像文件和设备树都准备好了，我们先学习如何通过网络启动 Linux，使用 tftp 命令将 zImage 下载到 DRAM 的 0X80800000 地址处，然后将设备树 imx6ull-alientek-emmc.dtb 下载到 DRAM 中的 0X83000000 地址处，最后之后命令 bootz 启动，命令如下：

```
tftp 80800000 zImage
tftp 83000000 imx6ull-alientek-emmc.dtb
bootz 80800000 – 83000000
```

命令运行结果如图 30.4.9.2 所示：

```
=> tftp 80800000 zImage
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##########################
         2.3 MiB/s
done
Bytes transferred = 6039328 (5c2720 hex)
=> tftp 83000000 imx6ull-alientek-emmc.dtb
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'imx6ull-alientek-emmc.dtb'.
Load address: 0x83000000
Loading: ###
         1.6 MiB/s
done
Bytes transferred = 37331 (91d3 hex)
=> bootz 80800000 - 83000000
Kernel image @ 0x80800000 [ 0x000000 - 0x5c2720 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300c1d2
Modify /soc/aips-bus@02100000/sim@0218c000:status disabled
ft_system_setup for mx6

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-2017.01) ) #9 SMP PREEMPT Fri Apr 19 17:15:22 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 UltraLite 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
Memory policy: Data cache writealloc
PERCPU: Embedded 12 pages/cpu @8bb31000 s17280 r8192 d23680 u49152
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
```

1、下载zImage
2、下载设备树imx6ull-alientek-emmc.dtb
3、使用命令“bootz”启动linux
4、Linux启动信息

图 30.4.9.2 通过网络启动 Linux

上图就是我们通过 tftp 和 bootz 命令来从网络启动 Linux 系统，如果我们要从 EMMC 中启动 Linux 系统的话只需要使用命令 fatload 将 zImage 和 imx6ull-alientek-emmc.dtb 从 EMMC 的分区 1 中拷贝到 DRAM 中，然后使用命令 bootz 启动即可。先使用命令 fatls 查看要下 EMMC 的分区 1 中有没有 Linux 镜像文件和设备树文件，如果没有的话参考 30.4.6 小节中讲解的 fatwrite 命令将 tftpboot 中的 zImage 和 imx6ull-alientek-emmc.dtb 文件烧写到 EMMC 的分区 1 中。然后

使用命令 fatload 将 zImage 和 imx6ull-alientek-emmc.dtb 文件拷贝到 DRAM 中，地址分别为 0X80800000 和 0X83000000，最后使用 bootz 启动，命令如下：

```
fatload mmc 1:1 80800000 zImage
fatload mmc 1:1 83000000 imx6ull-alientek-emmc.dtb
bootz 80800000 - 83000000
```

命令运行结果如图 30.4.9.3 所示：



图 30.4.6.3 从 EMMC 中启动 Linux

2、bootm 命令

bootm 和 bootz 功能类似，但是 bootm 用于启动 uImage 镜像文件。如果不使用设备树的话启动 Linux 内核的命令如下：

```
bootm addr
```

addr 是 uImage 镜像在 DRAM 中的首地址。

如果要使用设备树，那么 bootm 命令和 bootz 一样，命令格式如下：

```
bootm [addr [initrd[:size]] [fdt]]
```

其中 addr 是 uImage 在 DRAM 中的首地址，initrd 是 initrd 的地址，fdt 是设备树(.dtb)文件在 DRAM 中的首地址，如果 initrd 为空的话，同样是用“-”来替代。

3、boot 命令

boot 命令也是用来启动 Linux 系统的，只是 boot 会读取环境变量 bootcmd 来启动 Linux 系统，bootcmd 是一个很重要的环境变量！其名字分为“boot”和“cmd”，也就是“引导”和“命令”，说明这个环境变量保存着引导命令，其实就是启动的命令集合，具体的引导命令内容是可以修改的。比如我们要想使用 tftp 命令从网络启动 Linux 那么就可以设置 bootcmd 为“tftp 80800000 zImage; tftp 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000”，然后使用 saveenv 将 bootcmd 保存起来。然后直接输入 boot 命令即可从网络启动 Linux 系统，命令如下：

```
setenv bootcmd 'tftp 80800000 zImage; tftp 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000'
saveenv
boot
```

运行结果如图 30.4.6.4 所示：

```
=> setenv bootcmd 'tftp 80800000 zImage; tftp 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000'
=> saveenv
Saving Environment to MMC...
Writing to MMC(1)... done
=> boot
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ######################
         2.1 MiB/s
done
Bytes transferred = 6039328 (5c2720 hex)
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'imx6ull-alientek-emmc.dtb'.
Load address: 0x83000000
Loading: ###
         2.4 MiB/s
done
Bytes transferred = 37331 (91d3 hex)
Kernel image @ 0x80800000 [ 0x000000 - 0x5c2720 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300c1d2
Modify /soc/aips-bus@02100000/sim@0218c000:status disabled
ft_system_setup for mx6

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-2017.01) ) #9 SMP PREEMPT Fri Apr 19 17:15:22 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 UltraLite 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
```

1、设置环境变量bootcmd并保存
2、使用命令boot启动Linux
3、运行bootcmd中的启动命令启动Linux

图 30.4.6.4 设置 bootcmd 从网络启动 Linux

前面说过 uboot 倒计时结束以后就会启动 Linux 系统，其实就是执行的 bootcmd 中的启动命令。只要不修改 bootcmd 中的内容，以后每次开机 uboot 倒计时结束以后都会使用 tftp 命令从网络下载 zImage 和 imx6ull-alientek-emmc.dtb，然后启动 Linux。

如果想从 EMMC 启动那就设置 bootcmd 为“fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek_emmc.dtb; bootz 80800000 - 83000000”，然后使用 boot 命令启动即可，命令如下：

```
setenv bootcmd 'fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek_emmc.dtb; bootz 80800000 - 83000000'
savenev
boot
```

运行结果如图 30.4.6.5 所示：

```
=> setenv bootcmd 'fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek_emmc.dtb; bootz 80800000 - 83000000'
=> savenev
Unknown command 'savenev' - try 'help'
=> boot
reading zImage
6039328 bytes read in 132 ms (37.9 MiB/s)
reading imx6ull-alientek_emmc.dtb
** Unable to read file imx6ull-alientek_emmc.dtb **
Kernel image @ 0x80800000 [ 0x000000 - 0x5c2720 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   reserving fdt memory region: addr=83000000 size=a000
   Using Device Tree in place at 83000000, end 8300cfff
Modify /soc/aips-bus@02100000/sim@0218c000:status disabled
ft_system_setup for mx6

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-2017.01) ) #9 SMP PREEMPT Fri Apr 19 17:15:22 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 UltraLite 14x14 EVK Board
```

1、设置环境便令bootcmd并保存
2、启动Linux
3、启动过程

图 30.4.6.5 设置 bootcmd 从 EMMC 启动 Linux

如果不修改 bootcmd 的话，每次开机 uboot 倒计时结束以后都会自动从 EMMC 里面读取 zImage 和 imx6ull-alientek-emmc.dtb，然后启动 Linux。

30.4.10 其他常用命令

uboot 中还有其他一些常用的命令，比如 reset、go、run 和 mtest 等。

1、reset 命令

reset 命令顾名思义就是复位的，输入“reset”即可复位重启，如图 30.4.10.1 所示：

```
=> reset
resetting ...

U-Boot 2016.03 (Apr 21 2019 - 22:17:44 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 57C
Reset cause: WDOG
Board: MX6UL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT7016 (1024x600)
Video: 1024x600x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc1(part 0) is current device
Net:   FEC1
Normal Boot
Hit any key to stop autoboot:  0
=>
```

1、重启
2、重启后的uboot

图 30.4.10.1 reset 命令运行结果

2、go 命令

go 命令用于跳到指定的地址处执行应用，命令格式如下：

```
go addr [arg ...]
```

addr 是应用在 DRAM 中的首地址，我们可以编译一下裸机例程的实验 13_printf，然后将编译出来的 printf.bin 拷贝到 Ubuntu 中的 tftpboot 文件夹里面，注意，这里要拷贝 printf.bin 文件，不需要在前面添加 IVT 信息，因为 uboot 已经初始化好了 DDR 了。使用 tftp 命令将 printf.bin 下载到开发板 DRAM 的 0X87800000 地址处，因为裸机例程的链接首地址就是 0X87800000，最后使用 go 命令启动 printf.bin 这个应用，命令如下：

```
tftp 87800000 printf.bin
go 87800000
```

结果如图 30.4.10.2 所示：

```
=> tftp 87800000 printf.bin
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.50
Filename 'printf.bin'.
Load address: 0x87800000
Loading: #
         225.6 KiB/s
done
Bytes transferred = 11585 (2d41 hex)
=> go 87800000
## Starting application at 0x87800000 ...
输入两个整数，使用空格隔开:2 3
数据2 + 3 = 5

输入两个整数，使用空格隔开:54 6
数据54 + 6 = 60

输入两个整数，使用空格隔开:
```

1、下载printf.bin
2、运行printf.bin
3、printf.bin正常运行

图 30.4.10.2 go 命令运行裸机例程

从图 30.4.10.2 可以看出，通过 go 命令我们就可以在 uboot 中运行裸机例程。

3、run 命令

run 命令用于运行环境变量中定义的命令，比如可以通过"run bootcmd"来运行 bootcmd 中的启动命令，但是 run 命令最大的作用在于运行我们自定义的环境变量。在后面调试 Linux 系统的时候常常要在网络启动和 EMMC/NAND 启动之间来回切换，而 bootcmd 只能保存一种启动方式，如果要换另外一种启动方式的话就得重写 bootcmd，会很麻烦。这里我们就可以通过自定义环境变量来实现不同的启动方式，比如定义环境变量 mybootemmc 表示从 emmc 启动，定义 mybootnet 表示从网络启动，定义 mybootnand 表示从 NAND 启动。如果要切换启动方式的话只需要运行"run mybootxxx(xxx 为 emmc、net 或 nand)"即可。

说干就干，创建环境变量 mybootemmc、mybootnet 和 mybootnand，命令如下：

```
setenv mybootemmc 'fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek-emmc.dtb;bootz 80800000 - 83000000'
setenv mybootnand 'nand read 80800000 4000000 800000;nand read 83000000 6000000 100000;bootz 80800000 - 83000000'
setenv mybootnet 'tftp 80800000 zImage; tftp 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000'
saveenv
```

创建环境变量成功以后就可以使用 run 命令来运行 mybootemmc、mybootnet 或 mybootnand 来实现不同的启动：

```
run mybootemmc
```

或

```
run mytoobnand
```

或

```
run mybootnet
```

4、mtest 命令

metest 命令是一个简单的内存读写测试命令，可以用来测试自己开发板上的 DDR，命令格式如下：

```
mtest [start [end [pattern [iterations]]]]
```

start 是要测试的 DRAM 开始地址，end 是结束地址，比如我们测试 0X80000000~0X80001000 这段内存，输入"mtest 80000000 80001000"，结果如图 30.4.10.3 所示：

```
=> mtest 80000000 80001000
Testing 80000000 ... 80001000:
Pattern FFFFFFFF  Writing...    ading...Iteration:     486
```

图 30.4.10.3 mtest 命令运行结果

从图 30.4.10.3 可以看出，测试范围为 0X80000000~0X80001000，已经测试了 486 次，如果要结束测试就按下键盘上的"Ctrl+C"键。

至此，uboot 常用的命令就讲解完了，如果要使用 uboot 的其他命令，可以查看 uboot 中的帮助信息，或者上网查询一下相应的资料。

第三十一章 U-Boot 顶层 Makefile 详解

上一章我们详细的讲解了 uboot 的使用方法，其实就是各种命令的使用，学会 uboot 使用以后就可以尝试移植 uboot 到自己的开发板上了，但是在移植之前需要我们得先分析一遍 uboot 的启动流程源码，得捋一下 uboot 的启动流程，否则移植的时候都不知道该修改那些文件。本章我们就来分析一下正点原子提供的 uboot 源码，重点是分析 uboot 启动流程，而不是整个 uboot 源码，uboot 整个源码非常大，我们只看跟我们关心的部分即可。

31.1 U-Boot 工程目录分析

本书我们以 EMMC 版本的核心板为例讲解，为了方便，uboot 启动源码分析就在 Windows 下进行，将正点原子提供的 uboot 源码进行解压，解压完成以后的目录如图 31.1.1 所示：

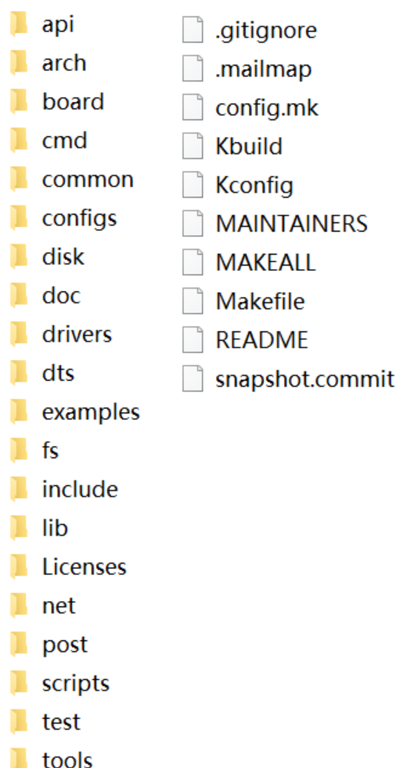


图 31.1.1 未编译的 uboot

图 31.1.1 是正点原子提供的未编译的 uboot 源码目录，我们在分析 uboot 源码之前一定要先在 Ubuntu 中编译一下 uboot 源码，因为编译过程会生成一些文件，而生成的这些恰恰是分析 uboot 源码不可或缺的文件。使用上一章创建的 shell 脚本来完成编译工作，命令如下：

```
cd alientek_uboot                                    //进入正点原子 uboot 源码目录
./mx6ull_alientek_emmc.sh                            //编译 uboot
cd ../                                               //返回上一级目录
tar -vcjf alientek_uboot.tar.bz2 alientek_uboot      //压缩
```

最终会生成一个名为 alientek_uboot.tar.bz2 的压缩包，将 alientek_uboot.tar.bz2 拷贝到 windows 系统中并解压，解压后的目录如图 31.1.2 所示：

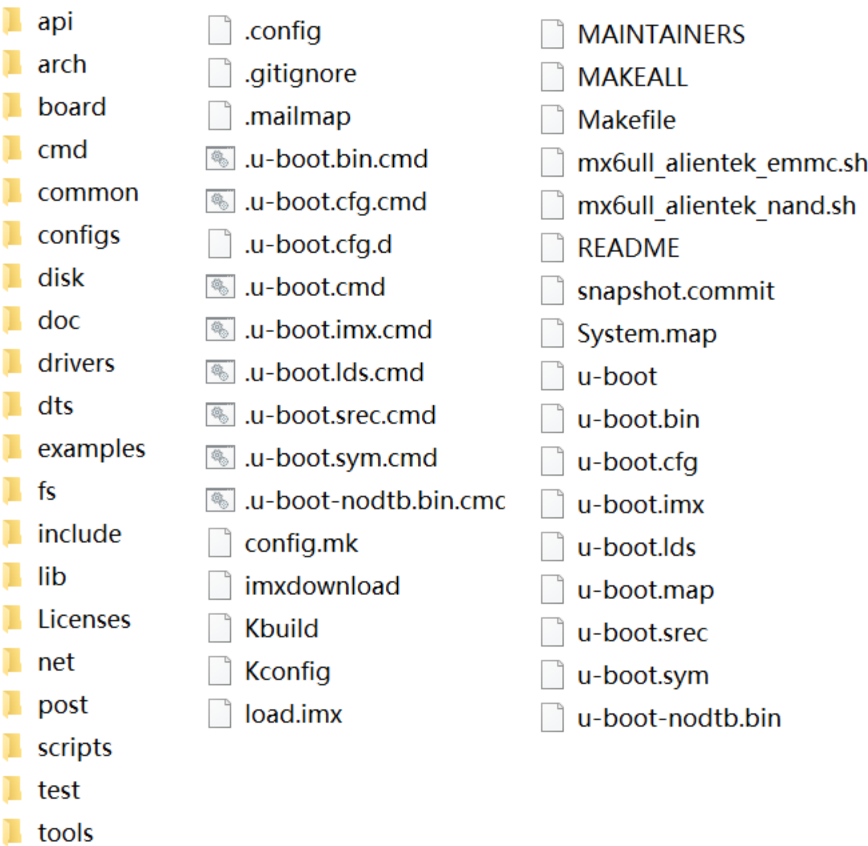


图 31.1.2 编译后的 uboot 源码文件

对比图 31.1.2 和图 31.1.1，可以看出编译后的 uboot 要比没编译之前多了好多文件，这些文件夹或文件的含义见表 31.1.1 所示：

类型	名字	描述	备注
文件夹	api	与硬件无关的 API 函数。	uboot 自带
	arch	与架构体系有关的代码。	
	board	不同板子(开发板)的定制代码。	
	cmd	命令相关代码。	
	common	通用代码。	
	configs	配置文件。	
	disk	磁盘分区相关代码	
	doc	文档。	
	drivers	驱动代码。	
	dts	设备树。	
	examples	示例代码。	
	fs	文件系统。	
	include	头文件。	
	lib	库文件。	
	Licenses	许可证相关文件。	
	net	网络相关代码。	

	post	上电自检程序。	
	scripts	脚本文件。	
	test	测试代码。	
	tools	工具文件夹。	
文件	.config	配置文件，重要的文件。	编译生成的文件
	.gitignore	git 工具相关文件。	uboot 自带
	.mailmap	邮件列表。	
	.u-boot.xxx.cmd (一系列)	这是一系列的文件，用于保存着一些命令。	编译生成的文件
	config.mk	某个 Makefile 会调用此文件。	uboot 自带
	imxdownload	正点原子编写的 SD 卡烧写软件。	正点原子提供
	Kbuild	用于生成一些和汇编有关的文件。	uboot 自带
	Kconfig	图形配置界面描述文件。	
	MAINTAINERS	维护者联系方式文件。	
	MAKEALL	一个 shell 脚本文件，帮助编译 uboot 的。	
	Makefile	主 Makefile，重要文件！	
	mx6ull_alientek_emmc.sh	上一章编写的编译脚本文件	上一章编写的。
	mx6ull_alientek_nand.sh	上一章编写的编译脚本文件	
	README	相当于帮助文档。	uboot 自带
	snapshot.commint	？？？	
	System.map	系统映射文件	编译出来的文件
	u-boot	编译出来的 u-boot 文件。	
	u-boot.xxx (一系列)	生成的一些 u-boot 相关文件，包括 u-boot.bin、u-boot.imx.等	

表 31.1.1 uboot 目录列表

表 31.1.1 中的很多文件夹和文件我们都不需要去关心，我们要关注的文件夹或文件如下：

1、arch 文件夹

这个文件夹里面存放着和架构有关的文件，如图 31.1.3 所示：

名称	修改日期	类型	大小
arc	2019-04-22 21:07	文件夹	
arm	2019-04-22 21:07	文件夹	
avr32	2019-04-22 21:07	文件夹	
blackfin	2019-04-22 21:07	文件夹	
m68k	2019-04-22 21:07	文件夹	
microblaze	2019-04-22 21:07	文件夹	
mips	2019-04-22 21:07	文件夹	
nds32	2019-04-22 21:07	文件夹	
nios2	2019-04-22 21:07	文件夹	
openrisc	2019-04-22 21:07	文件夹	
powerpc	2019-04-22 21:07	文件夹	
sandbox	2019-04-22 21:07	文件夹	
sh	2019-04-22 21:07	文件夹	
sparc	2019-04-22 21:07	文件夹	
x86	2019-04-22 21:07	文件夹	
.gitignore	2019-03-26 15:49	GITIGNORE 文件	1 KB
Kconfig	2019-03-26 15:49	文件	5 KB

图 31.1.3 arch 文件夹

从图 31.1.3 可以看出有很多架构，比如 arm、avr32、m68k 等，我们现在用的是 ARM 芯片，所以只需要关心 arm 文件夹即可，打开 arm 文件夹里面内容如图 31.1.4 所示：

名称	修改日期	类型
cpu	2019-04-22 21:07	文件夹
dts	2019-04-22 21:07	文件夹
imx-common	2019-04-22 21:07	文件夹
include	2019-04-22 21:07	文件夹
lib	2019-04-22 21:07	文件夹
mach-at91	2019-04-22 21:07	文件夹
mach-bcm283x	2019-04-22 21:07	文件夹
mach-davinci	2019-04-22 21:07	文件夹
mach-exynos	2019-04-22 21:07	文件夹
mach-highbank	2019-04-22 21:07	文件夹
mach-integrator	2019-04-22 21:07	文件夹
mach-keystone	2019-04-22 21:07	文件夹

图 31.1.4 arm 文件夹

图 31.1.4 只截取了一部分，还有一部分 mach-xxx 的文件夹。mach 开头的文件夹是跟具体的设备有关的，比如“mach-exynos”就是跟三星的 exyons 系列 CPU 有关的文件。我们使用的是 I.MX6ULL，所以要关注“imx-common”这个文件夹。另外“cpu”这个文件夹也是和 cpu 架构有关的，打开以后如图 31.1.5 所示：

名称	日期	类型	大小
arm11	2019-04-22 21:07	文件夹	
arm720t	2019-04-22 21:07	文件夹	
arm920t	2019-04-22 21:07	文件夹	
arm926ejs	2019-04-22 21:07	文件夹	
arm946es	2019-04-22 21:07	文件夹	
arm1136	2019-04-22 21:07	文件夹	
arm1176	2019-04-22 21:07	文件夹	
armv7	2019-04-22 21:07	文件夹	
armv7m	2019-04-22 21:07	文件夹	
armv8	2019-04-22 21:07	文件夹	
pxa	2019-04-22 21:07	文件夹	
sa1100	2019-04-22 21:07	文件夹	
.built-in.o.cmd	2019-04-22 20:52	Windows 命令脚本	1 KB
built-in.o	2019-04-22 20:52	O 文件	1 KB
Makefile	2019-03-26 15:49	文件	1 KB
u-boot.lds	2019-03-26 15:49	LDS 文件	3 KB
u-boot-spl.lds	2019-03-26 15:49	LDS 文件	2 KB

图 31.1.5 cpu 文件夹

从图 31.1.5 可以看出有多种 ARM 架构相关的文件夹，I.MX6ULL 使用的 Cortex-A7 内核，Cortex-A7 属于 armv7，所以我们要关心“armv7”这个文件夹。cpu 文件夹里面有个名为就“u-boot.lds”的链接脚本文件，这个就是 ARM 芯片所使用的 u-boot 链接脚本文件！armv7 这个文件夹里面的文件都是跟 ARMV7 架构有关的，是我们分析 uboot 启动源码的时候需要重点关注的。

2、board 文件夹

board 文件夹就是和具体的板子有关的，打开此文件夹，里面全是不同的板子，毫无疑问！正点原子的开发板肯定也在里面(正点原子添加的)，borad 文件夹里面有个名为“freescale”的文件夹，如图 31.1.6 所示：

名称	日期	类型
evb_rk3036	2019-04-22 21:07	文件夹
firefly	2019-04-22 21:07	文件夹
freescale	2019-04-22 21:07	文件夹
gaisler	2019-04-22 21:07	文件夹

图 31.1.6 freescale 文件夹

所有使用 freescale 芯片的板子都放到此文件夹中，I.MX 系列以前属于 freescale，只是 freescale 后来被 NXP 收购了。打开此 freescale 文件夹，在里面找到和 mx6u(I.MX6UL/ULL)有关的文件夹，如图 31.1.7 所示：

名称	日期	类型
mx6ul_14x14_ddr3_arm2	2019-09-03 0:17	文件夹
mx6ul_14x14_evk	2019-09-03 0:17	文件夹
mx6ul_14x14_lpddr2_arm2	2019-09-03 0:17	文件夹
mx6ull_ddr3_arm2	2019-09-03 0:17	文件夹
mx6ullevk	2019-09-03 0:17	文件夹

图 31.1.7 mx6u 相关板子

图 31.1.7 中有 5 个文件夹，这 5 个文件夹对应 5 种板子，以“mx6ul”开头的表示使用 I.MX6UL 芯片的板子，以 mx6ull 开头的表示使用 I.MX6ULL 芯片的板子。mx6ullevk 是 NXP 官方的 I.MX6ULL 开发板，正点原子的 ALPHA 开发板就是在这个基础上开发的，因此 mx6ullevk 也是正点原子的开发板。我们后面移植 uboot 到时候就是参考的 NXP 官方的开发板，也就是要

参考 mx6ullevk 这个文件夹来定义我们的板子。

3、configs 文件夹

此文件夹为 uboot 配置文件，uboot 是可配置的，但是你要是自己从头开始一个一个项目的配置，那就太麻烦了，因此一般半导体或者开发板厂商都会制作好一个配置文件。我们可以在这个做好的配置文件基础上来添加自己想要的功能，这些半导体厂商或者开发板厂商制作好的配置文件统一命名为"xxx_defconfig"，xxx 表示开发板名字，这些 defconfig 文件都存放在 configs 文件夹，因此，NXP 官方开发板和正点原子的开发板配置文件肯定也在这个文件夹中，如图 31.1.8 所示：

mx6ull_14x14_ddr3_arm2_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_emmc_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_epdc_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_nand_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_qspi1_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_spinor_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr3_arm2_tsc_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr256_emmc_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr256_nand_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr256_nand_sd_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr512_emmc_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr512_nand_defconfig	2019-08-31 11:46	文件	1 KB
mx6ull_14x14_ddr512_nand_sd_defconfig	2019-08-31 11:46	文件	1 KB

正点原子ALPHA开发板对应的默认配置文件

图 31.1.8 正点原子开发板配置文件

图 31.1.8 中这 6 个文件就是正点原子 I.MX6U-ALPHA 开发板所对应的 uboot 默认配置文件。我们只关心 mx6ull_14x14_ddr512_emmc_defconfig 和 mx6ull_14x14_ddr256_nand_defconfig 这两根文件，分别是正点原子 I.MX6ULL EMMC 核心板和 NAND 核心板的配置文件。使用 "make xxx_defconfig" 命令即可配置 uboot，比如：

```
make mx6ull_14x14_ddr512_emmc_defconfig
```

上述命令就是配置正点原子的 I.MX6ULL EMMC 核心板所使用的 uboot。

在编译 uboot 之前一定要使用 defconfig 来配置 uboot！

在编译 uboot 之前一定要使用 defconfig 来配置 uboot！

在编译 uboot 之前一定要使用 defconfig 来配置 uboot！

在 mx6ull_alientek_emmc.sh 中就有下面这一句：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_ddr512_emmc_defconfig
```

这个就是调用 mx6ull_14x14_ddr512_emmc_defconfig 来配置 uboot，只是这个命令还带了一些其它参数而已。

4、.u-boot.xxx_cmd 文件

.u-boot.xxx_cmd 是一系列的文件，这些文件都是编译生成的，都是一些命令文件，比如文件.u-boot.bin.cmd，看名字应该是和 u-boot.bin 有关的，此文件的内容如下：

示例代码 31.1.1 .u-boot.bin.cmd 代码

```
1  cmd_u-boot.bin := cp u-boot-nodtb.bin u-boot.bin
```

.u-boot.bin.cmd 里面定义了一个变量：cmd_u-boot.bin，此变量的值为 "cp u-boot-nodtb.bin u-boot.bin"，也就是拷贝一封 u-boot-nodtb.bin 文件，并且重命名为 u-boot.bin，这个就是 u-boot.bin 的来源，来自于文件 u-boot-nodtb.bin。

那么 u-boot-nodtb.bin 是怎么来的呢？文件.u-boot-nodtb.bin.cmd 就是用于生成 u-boot.nodtb.bin 的，此文件内容如下：

示例代码 31.1.2 .u-boot-nodtb.bin.cmd 代码

```
1 cmd_u-boot-nodtb.bin := arm-linux-gnueabihf-objcopy --gap-fill=0xff -j .text -j .secure_text -j .rodata -j .hash -j .data -j .got -j .got.plt -j .u_boot_list -j .rel.dyn -O binary  u-boot u-boot-nodtb.bin
```

这里用到了 arm-linux-gnueabihf-objcopy，使用 objcopy 将 ELF 格式的 u-boot 文件转换为二进制的 u-boot-nodtb.bin 文件。

文件 u-boot 是 ELF 格式的文件，文件.u-boot.cmd 用于生成 u-boot，文件内容如下：

示例代码 31.1.3 .u-boot.cmd 代码

```
1 cmd_u-boot := arm-linux-gnueabihf-ld.bfd   -pie  --gc-sections -Bstatic -Ttext 0x87800000 -o u-boot -T u-boot.lds arch/arm/cpu/armv7/start.o --start-group  arch/arm/cpu/built-in.o arch/arm/cpu/armv7/built-in.o  arch/arm/imx-common/built-in.o arch/arm/lib/built-in.o  board/freescale/common/built-in.o board/freescale/mx6ull_alientek_emmc/built-in.o  cmd/built-in.o common/built-in.o  disk/built-in.o  drivers/built-in.o drivers/dma/built-in.o  drivers/gpio/built-in.o  drivers/i2c/built-in.o  drivers/mmc/built-in.o  drivers/mtd/built-in.o drivers/mtd/onenand/built-in.o  drivers/mtd/spi/built-in.o drivers/net/built-in.o  drivers/net/phy/built-in.o  drivers/pci/built-in.o  drivers/power/built-in.o  drivers/power/battery/built-in.o drivers/power/fuel_gauge/built-in.o  drivers/power/mfd/built-in.o drivers/power/pmic/built-in.o  drivers/power/regulator/built-in.o drivers/serial/built-in.o  drivers/spi/built-in.o drivers/usb/dwc3/built-in.o  drivers/usb/emul/built-in.o drivers/usb/eth/built-in.o  drivers/usb/gadget/built-in.o drivers/usb/gadget/udc/built-in.o  drivers/usb/host/built-in.o drivers/usb/musb-new/built-in.o  drivers/usb/musb/built-in.o drivers/usb/phy/built-in.o  drivers/usb/ulpi/built-in.o  fs/built-in.o lib/built-in.o  net/built-in.o  test/built-in.o  test/dm/built-in.o --end-group arch/arm/lib/eabi_compat.o  -L /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/4.9.4 -lgcc -Map u-boot.map
```

.u-boot.cmd 使用到了 arm-linux-gnueabihf-ld.bfd，也就是链接工具，使用 ld.bfd 将各个 built-in.o 文件链接在以前就形成了 u-boot 文件。uboot 在编译的时候会将同一个目录中的所有.c 文件都编译在一起，并命名为 built-in.o，相当于将众多的.c 文件对应的.o 文件集合在一起，这个就是 u-boot 文件的来源。

如果我们要用 NXP 提供的 MFGTools 工具向开发板烧写 uboot，此时烧写的是 u-boot.imx 文件，而不是 u-boot.bin 文件。u-boot.imx 是在 u-boot.bin 文件的头部添加了 IVT、DCD 等信息。这个工作是由文件.u-boot.imx.cmd 来完成的，此文件内容如下：

示例代码 31.1.4 .u-boot.imx.cmd 代码

```
1 cmd_u-boot.imx := ./tools/mkimage -n
  board/freescale/mx6ull_alientek_emmc/imximage.cfg.cfgtmp -T imximage -
  e 0x87800000 -d u-boot.bin u-boot.imx
```

可以看出，这里用到了工具 tools/mkimage，而 IVT、DCD 等数据保存在了文件 board/freescale/mx6ullevk/imximage-ddr512.cfg.cfgtmp 中(如果是 NAND 核心板的话就是 imximage-ddr256.cfg.cfgtmp)，工具 mkimage 就是读取文件 imximage-ddr512.cfg.cfgtmp 里面的信息，然后将其添加到文件 u-boot.bin 的头部，最终生成 u-boot.imx。

文件.u-boot.lds.cmd 就是用于生成 u-boot.lds 链接脚本的，由于.u-boot.lds.cmd 文件内容太多，这里就不列出来了。uboot 根目录下的 u-boot.lds 链接脚本就是来源于 arch/arm/cpu/u-boot.lds 文件。

还有一些其它的.u-boot.lds.xxx.cmd 文件，大家自行分析一下，关于.u-boot.lds.xxx.cmd 文件就讲解到这里。

5、Makefile 文件

这个是顶层 Makefile 文件，Makefile 是支持嵌套的，也就是顶层 Makefile 可以调用子目录中的 Makefile 文件。Makefile 嵌套在大项目中很常见，一般大项目里面所有的源代码都不会放到同一个目录中，各个功能模块的源代码都是分开的，各自存放在各自的目录中。每个功能模块目录下都有一个 Makefile，这个 Makefile 只处理本模块的编译链接工作，这样所有的编译链接工作就不用全部放到一个 Makefile 中，可以使得 Makefile 变得简洁明了。

uboot 源码根目录下的 Makefile 是顶层 Makefile，他会调用其它的模块的 Makefile 文件，比如 drivers/adc/Makefile。当然了，顶层 Makefile 要做的工作可远不止调用子目录 Makefile 这么简单，关于顶层 Makefile 的内容我们稍后会有详细的讲解。

6、u-boot.xxx 文件

u-boot.xxx 同样也是一系列文件，包括 u-boot、u-boot.bin、u-boot.cfg、u-boot.imx、u-boot.lds、u-boot.map、u-boot.srec、u-boot.sym 和 u-boot-nodtb.bin，这些文件的含义如下：

u-boot：编译出来的 ELF 格式的 uboot 镜像文件。

u-boot.bin：编译出来的二进制格式的 uboot 可执行镜像文件。

u-boot.cfg：uboot 的另外一种配置文件。

u-boot.imx：u-boot.bin 添加头部信息以后的文件，NXP 的 CPU 专用文件。

u-boot.lds：链接脚本。

u-boot.map：uboot 映射文件，通过查看此文件可以知道某个函数被链接到了哪个地址上。

u-boot.srec：S-Record 格式的镜像文件。

u-boot.sym：uboot 符号文件。

u-boot-nodtb.bin：和 u-boot.bin 一样，u-boot.bin 就是 u-boot-nodtb.bin 的复制文件。

7、.config 文件

uboot 配置文件，使用命令“make xxx_defconfig”配置 uboot 以后就会自动生成，.config 内容如下：

示例代码 31.1.5 .config 代码

```
1 #
2 # Automatically generated file; DO NOT EDIT.
3 # U-Boot 2016.03 Configuration
4 #
5 CONFIG_CREATE_ARCH_SYMLINK=y
```

```
6  CONFIG_HAVE_GENERIC_BOARD=y
7  CONFIG_SYS_GENERIC_BOARD=y
8  # CONFIG_ARC is not set
9  CONFIG_ARM=y
10 # CONFIG_AVR32 is not set
11 # CONFIG_BLACKFIN is not set
12 # CONFIG_M68K is not set
13 # CONFIG_MICROBLAZE is not set
14 # CONFIG_MIPS is not set
15 # CONFIG_NDS32 is not set
16 # CONFIG_NIOS2 is not set
17 # CONFIG_OPENRISC is not set
18 # CONFIG_PPC is not set
19 # CONFIG_SANDBOX is not set
20 # CONFIG_SH is not set
21 # CONFIG_SPARC is not set
22 # CONFIG_X86 is not set
23 CONFIG_SYS_ARCH="arm"
24 CONFIG_SYS_CPU="armv7"
25 CONFIG_SYS_SOC="mx6"
26 CONFIG_SYS_VENDOR="freescale"
27 CONFIG_SYS_BOARD="mx6ull_alientek_emmc"
28 CONFIG_SYS_CONFIG_NAME="mx6ull_alientek_emmc"
29
30 ......
31
32 #
33 # Boot commands
34 #
35 CONFIG_CMD_BOOTD=y
36 CONFIG_CMD_BOOTM=y
37 CONFIG_CMD_ELF=y
38 CONFIG_CMD_GO=y
39 CONFIG_CMD_RUN=y
40 CONFIG_CMD_IMI=y
41 CONFIG_CMD_IMLS=y
42 CONFIG_CMD_XIMG=y
43
44 #
45 # Environment commands
46 #
47 CONFIG_CMD_EXPORTENV=y
48 CONFIG_CMD_IMPORTENV=y
```

```
49 CONFIG_CMD_EDITENV=y
50 CONFIG_CMD_SAVEENV=y
51 CONFIG_CMD_ENV_EXISTS=y
52
53 ......
54
55 #
56 # Library routines
57 #
58 # CONFIG_CC_OPTIMIZE_LIBS_FOR_SPEED is not set
59 CONFIG_HAVE_PRIVATE_LIBGCC=y
60 # CONFIG_USE_PRIVATE_LIBGCC is not set
61 CONFIG_SYS_HZ=1000
62 # CONFIG_USE_TINY_PRINTF is not set
63 CONFIG_REGEX=y
```

可以看出.config 文件中都是以“CONFIG_”开始的配置项，这些配置项就是 Makefile 中的变量，因此后面都跟有相应的值，uboot 的顶层 Makefile 或子 Makefile 会调用这些变量值。在.config 中会有大量的变量值为‘y’，这些为‘y’的变量一般用于控制某项功能是否使能，为‘y’的话就表示功能使能，比如：

```
CONFIG_CMD_BOOTD=y
```

如果使能了 bootd 这个命令的话，CONFIG_CMD_BOOTM 就为‘y’。在 cmd/Makefile 中有如下代码：

示例代码 31.1.6 cmd/Makefile 代码

```
1 ifndef CONFIG_SPL_BUILD
2 # core command
3 obj-y += boot.o
4 obj-$(CONFIG_CMD_BOOTM) += bootm.o
5 obj-y += help.o
6 obj-y += version.o
```

在示例代码 31.1.6 中，有如下所示一行代码：

```
obj-$(CONFIG_CMD_BOOTM) += bootm.o
```

CONFIG_CMD_BOOTM=y，将其展开就是：

```
obj-y += bootm.o
```

也就是给 obj-y 追加了一个“bootm.o”，obj-y 包含着所有要编译的文件对应的.o 文件，这里表示需要编译文件 cmd/bootm.c。相当于通过“CONFIG_CMD_BOOTD=y”来使能 bootm 这个命令，进而编译 cmd/bootm.c 这个文件，这个文件实现了命令 bootm。在 uboot 和 Linux 内核中都是采用这种方法来选择使能某个功能，编译对应的源码文件。

8、README

README 文件描述了 uboot 的详细信息，包括 uboot 该如何编译、uboot 中各文件夹的含义、相应的命令等等。建议大家详细的阅读此文件，可以进一步增加对 uboot 的认识。

关于 uboot 根目录中的文件和文件夹的含义就讲解到这里，接下来就要开始分析 uboot 的启动流程了。

31.2 VScode 工程创建

先在 Ubuntu 下编译一下 uboot，然后将编译后的 uboot 文件夹复制到 windows 下，并创建 VScode 工程。打开 VScode，选择：文件->打开文件夹…，选中 uboot 文件夹，如图 31.2.1 所示：

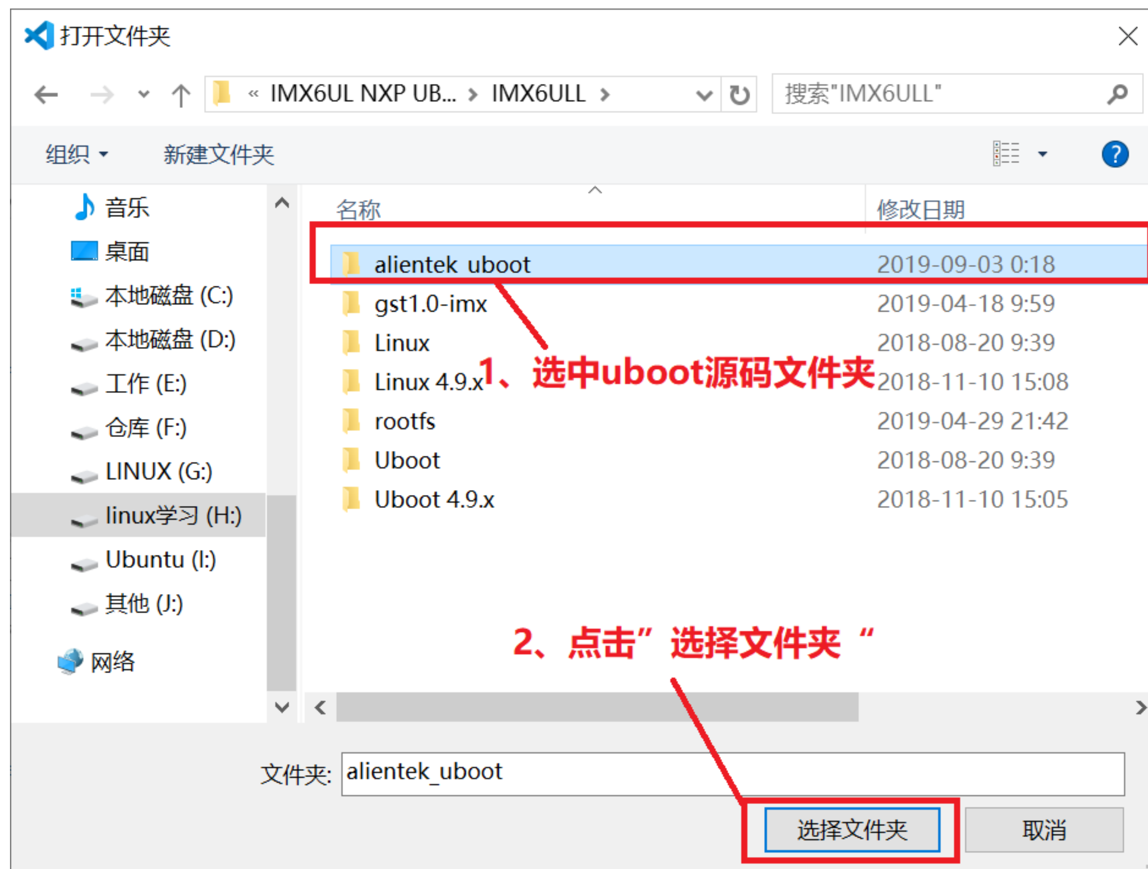


图 31.2.1 选择 uboot 源码文件夹

打开 uboot 目录以后，VSCode 界面如图 31.2.2 所示：



图 31.2.2 VScode 界面

点击“文件->将工作区另存为...”，打开保存工作区对话框，将工作区保存到 uboot 源码根目录下，设置文件名为“uboot”，如图 31.2.3 所示：

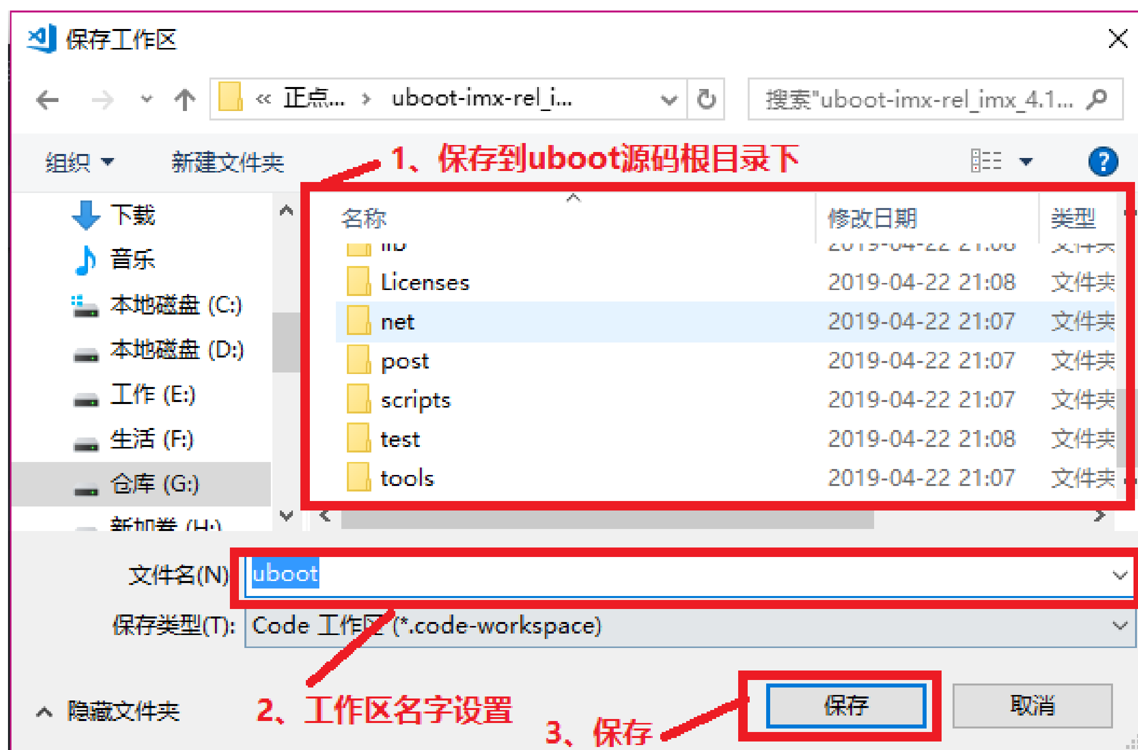


图 31.2.3 保存工作区

保存成功以后就会在 uboot 源码根目录下存在一个名为 uboot.code-workspace 的文件。这样一个完整的 VSCode 工程就建立起来了。但是这个 VSCode 工程包含了 uboot 的所有文件，uboot 中有些文件是不需要的，比如 arch 目录下是各种架构的文件夹，如图 31.2.4 所示：

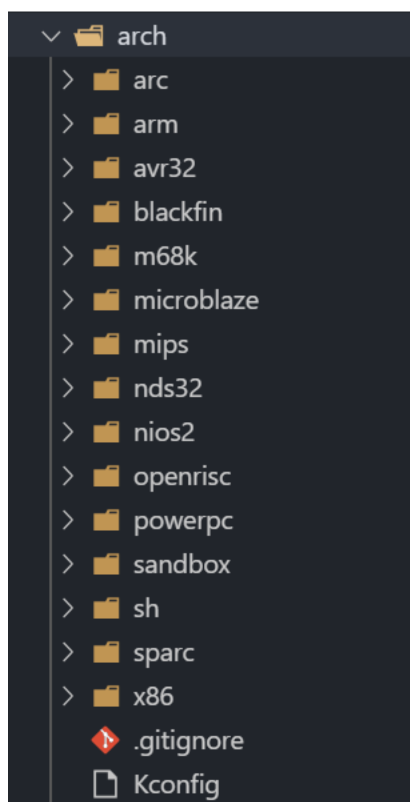


图 31.2.4 arch 目录

在 arch 目录下，我们只需要 arm 文件夹，所以需要将其它的目录从 VSCode 中给屏蔽掉，比如将 arch/avr32 这个目录给屏蔽掉。

在 VSCode 上建名为“.vscode”的文件夹，如图 31.2.5 所示：



图 31.2.5 新建.vscode 文件夹

输入新建文件夹的名字，完成以后如图 31.2.6 所示。

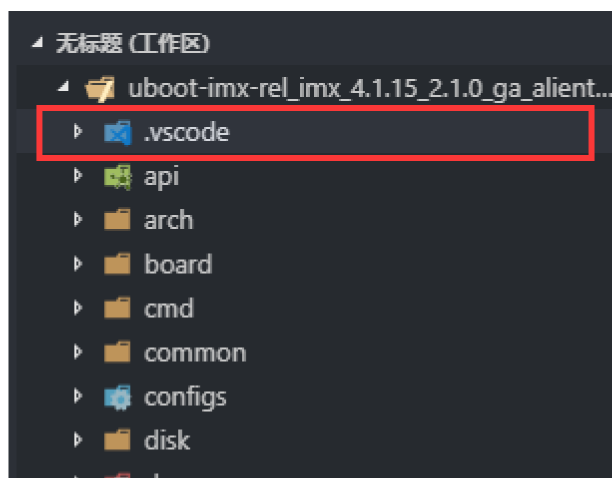


图 31.2.6 新建的.vscode 文件夹

在.vscode 文件夹中新建一个名为“settings.json”的文件，然后在 settings.json 中输入如下内容：

示例代码 31.2.1settings.json 文件代码

```
1  {
2      "search.exclude": {
3          "**/node_modules": true,
4          "**/bower_components": true,
5      },
6      "files.exclude": {
7          "**/.git": true,
8          "**/.svn": true,
9          "**/.hg": true,
10         "**/CVS": true,
11         "**/.DS_Store": true,
12     }
13 }
```

结果如图 31.2.7 所示：

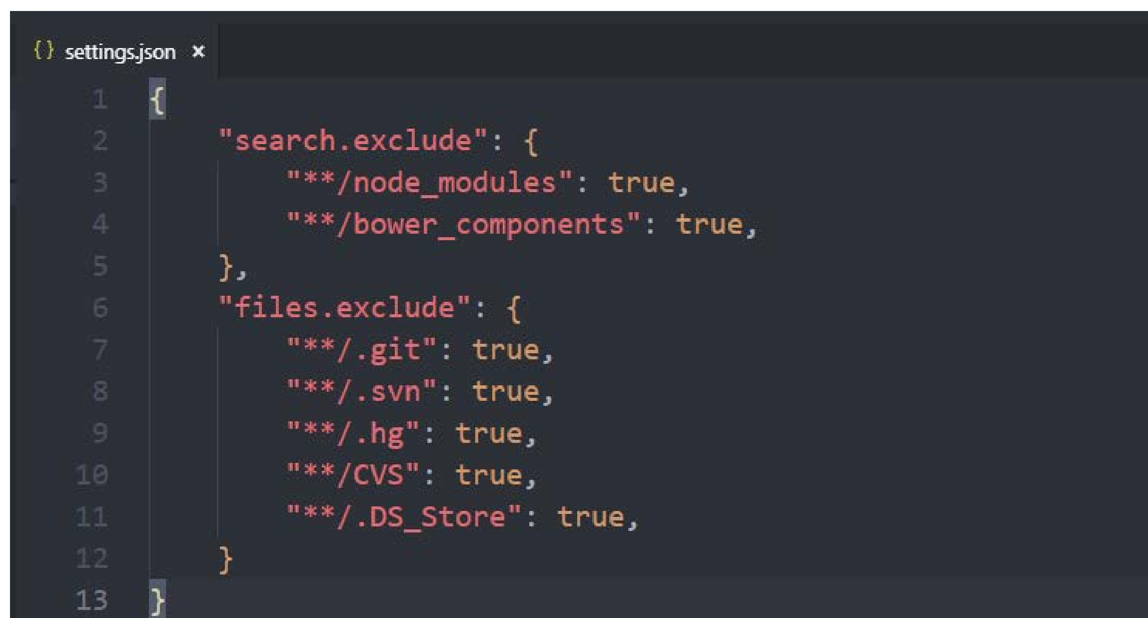


图 31.2.7 settings.json 文件默认内容

其中"search.exclude"里面是需要在搜索结果中排除的文件或者文件夹，"files.exclude"是左侧工程目录中需要排除的文件或者文件夹。我们需要将 arch/avr32 文件夹下的所有文件从搜索结果和左侧的工程目录中都排除掉，因此在"search.exclude"和"files.exclude"中输入如图 31.2.8 所示内容：

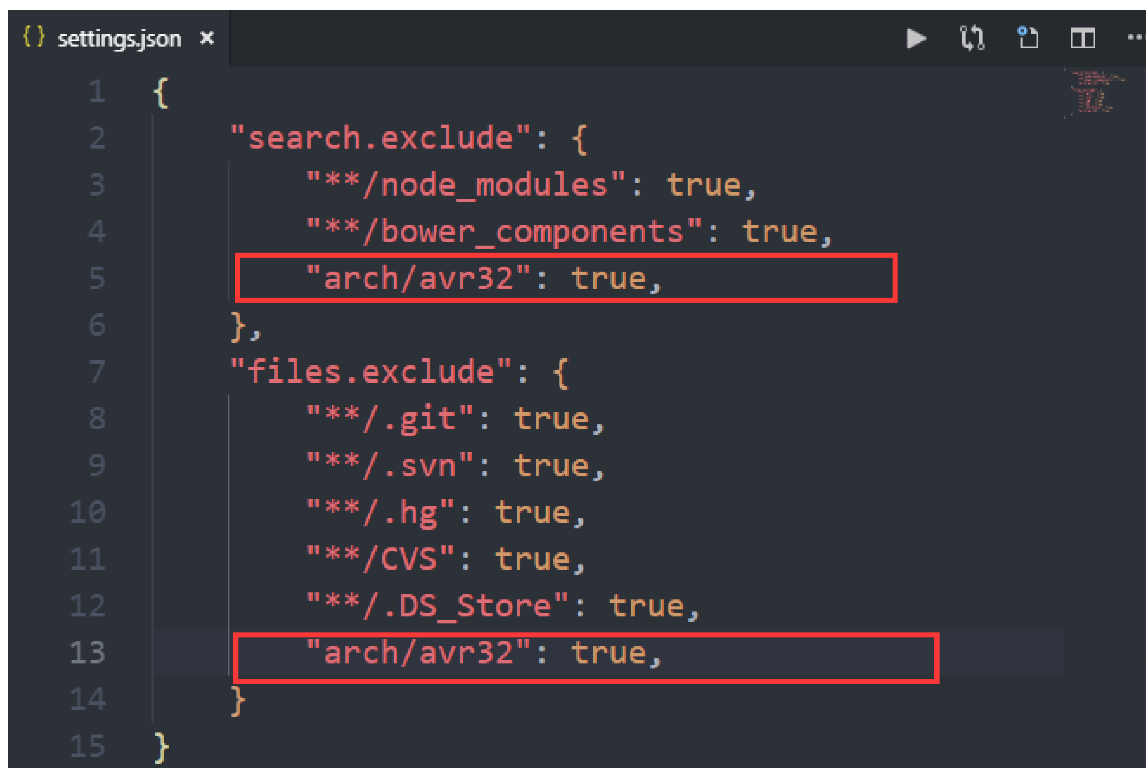


图 31.2.8 排除 arch/avr32 目录

此时再看一下左侧的工程目录，发现 arch 目录下没有 avr32 这个文件夹了，说明 avr32 这个文件夹被排除掉了，如图 31.2.9 所示：

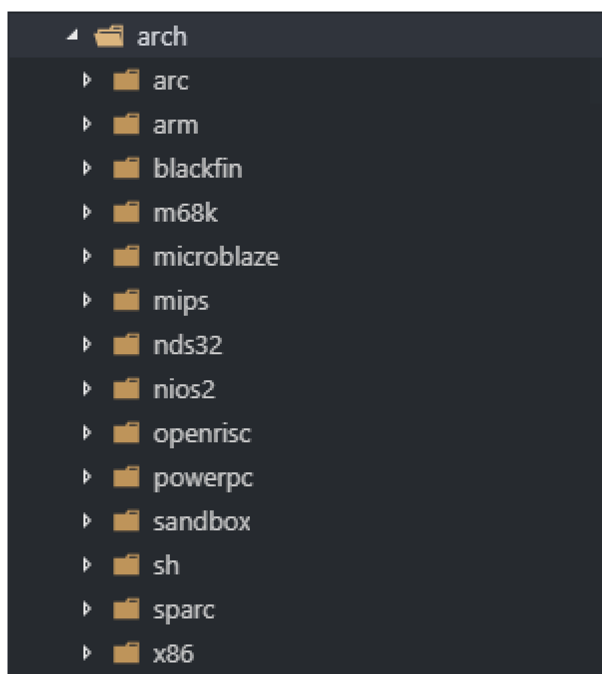


图 31.2.9 arch/avr32 目录排除

我们只是在"search.exclude"和"files.exclude"中加入了：arch/avr32": true，冒号前面的是要排除的文件或者文件夹，冒号后面为是否将文件排除，true 表示排除，false 表示不排除。用这种方法即可将不需要的文件，或者文件夹排除掉，对于本章我们分析 uboot 而言，在"search.exclude"

和"files.exclude"中需要输入的完成的内容如下：

示例代码 31.2.2 settings.json 文件代码

```
1  "**/*.o":true,
2  "**/*.su":true,
3  "**/*.cmd":true,
4  "arch/arc":true,
5  "arch/avr32":true,
6  "arch/blackfin":true,
7  "arch/m68k":true,
8  "arch/microblaze":true,
9  "arch/mips":true,
10 "arch/nds32":true,
11 "arch/nios2":true,
12 "arch/openrisc":true,
13 "arch/powerpc":true,
14 "arch/sandbox":true,
15 "arch/sh":true,
16 "arch/sparc":true,
17 "arch/x86":true,
18 "arch/arm/mach*":true,
19 "arch/arm/cpu/arm11*":true,
20 "arch/arm/cpu/arm720t":true,
21 "arch/arm/cpu/arm9*":true,
22 "arch/arm/cpu/armv7m":true,
23 "arch/arm/cpu/armv8":true,
24 "arch/arm/cpu/pxa":true,
25 "arch/arm/cpu/sa1100":true,
26 "board/[a-e]*":true,
27 "board/[g-z]*":true,
28 "board/[0-9]*":true,
29 "board/[A-Z]*":true,
30 "board/fir*":true,
31 "board/freescale/b*":true,
32 "board/freescale/l*":true,
33 "board/freescale/m5*":true,
34 "board/freescale/mp*":true,
35 "board/freescale/c29*":true,
36 "board/freescale/cor*":true,
37 "board/freescale/mx7*":true,
38 "board/freescale/mx2*":true,
39 "board/freescale/mx3*":true,
40 "board/freescale/mx5*":true,
41 "board/freescale/p*":true,
```

```
42 "board/freescale/q*":true,
43 "board/freescale/t*":true,
44 "board/freescale/v*":true,
45 "configs/[a-l]*":true,
46 "configs/[n-z]*":true,
47 "configs/[A-Z]*":true,
48 "configs/M[a-z]*":true,
49 "configs/M[A-Z]*":true,
50 "configs/M[0-9]*":true,
51 "configs/m[a-w]*":true,
52 "configs/m[0-9]*":true,
53 "configs/[0-9]*":true,
54 "include/configs/[a-l]*":true,
55 "include/configs/[n-z]*":true,
56 "include/configs/[A-Z]*":true,
57 "include/configs/m[a-w]*":true,
```

上述代码用到了通配符"*"，比如"**/*.o"表示所有.o 结尾的文件。"configs/[a-l]*"表示 configs 目录下所有以'a'～'l'开头的文件或者文件夹。上述配置只是排除了一部分文件夹，大家在实际的使用中可以根据自己的实际需求来选择将哪些文件或者文件夹排除掉。排除以后我们的工程就会清爽很多，搜索的时候也不会跳出很多文件了。

31.3 U-Boot 顶层 Makefile 分析

在阅读 uboot 源码之前，肯定是要先看一下顶层 Makefile，分析 gcc 版本代码的时候一定是先从顶层 Makefile 开始的，然后再是子 Makefile，这样通过层层分析 Makefile 即可了解整个工程的组织结构。顶层 Makefile 也就是 uboot 根目录下的 Makefile 文件，由于顶层 Makefile 文件内容比较多，所以我们将其分开来看。

31.3.1 版本号

顶层 Makefile 一开始是版本号，内容如下(为了方便分析，顶层 Makefile 代码段前段行号采用 Makefile 中的行号，因为 uboot 会更新，因此行号可能会与你所看的顶层 Makefile 有所不同)：

示例代码 31.3.1.1 顶层 Makefile 代码

```
5 VERSION = 2016
6 PATCHLEVEL = 03
7 SUBLEVEL =
8 EXTRAVERSION =
9 NAME =
```

VERSION 是主版本号，PATCHLEVEL 是补丁版本号，SUBLEVEL 是次版本号，这三个一起构成了 uboot 的版本号，比如当前的 uboot 版本号就是"2016.03"。EXTRAVERSION 是附加版本信息，NAME 是和名字有关的，一般不使用这两个。

31.3.2 MAKEFLAGS 变量

make 是支持递归调用的，也就是在 Makefile 中使用“make”命令来执行其他的 Makefile 文件，一般都是子目录中的 Makefile 文件。假如在当前目录下存在一个“subdir”子目录，这个子目录中又有其对应的 Makefile 文件，那么这个工程在编译的时候其主目录中的 Makefile 就可以调用子目录中的 Makefile，以此来完成所有子目录的编译。主目录的 Makefile 可以使用如下代码来编译这个子目录：

```
$(MAKE) -C subdir
```

$(MAKE)就是调用“make”命令，-C 指定子目录。有时候我们需要向子 make 传递变量，这个时候使用“export”来导出要传递给子 make 的变量即可，如果不希望哪个变量传递给子 make 的话就使用“unexport”来声明不导出:

```
export VARIABLE ……            //导出变量给子 make 。
unexport VARIABLE……            //不导出变量给子 make。
```

有两个特殊的变量：“SHELL”和“MAKEFLAGS”，这两个变量除非使用“unexport”声明，否则的话在整个make的执行过程中，它们的值始终自动的传递给子make。在uboot的主Makefile中有如下代码：

示例代码 31.3.2.1 顶层 Makefile 代码

```
20 MAKEFLAGS += -rR --include-dir=$(CURDIR)
```

上述代码使用“+=”来给变量 MAKEFLAGS 追加了一些值，“-rR”表示禁止使用内置的隐含规则和变量定义，“--include-dir”指明搜索路径，”$(CURDIR)”表示当前目录。

31.3.3 命令输出

uboot 默认编译是不会在终端中显示完整的命令，都是短命令，如图 31.3.3 所示：

```
  CC      examples/standalone/stubs.o
  LD      examples/standalone/libstubs.o
  CC      examples/standalone/hello_world.o
  LD      examples/standalone/hello_world
  OBJCOPY examples/standalone/hello_world.srec
  OBJCOPY examples/standalone/hello_world.bin
  LDS     u-boot.lds
  LD      u-boot
  OBJCOPY u-boot-nodtb.bin
  COPY    u-boot.bin
  CFGS    board/freescale/mx6ull_alientek_emmc/imximage.cfg.cfgtmp
  MKIMAGE u-boot.imx
  OBJCOPY u-boot.srec
  SYM     u-boot.sym
  CFG     u-boot.cfg
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 31.3.3.1 终端短命令输出

在终端中输出短命令虽然看起来很清爽，但是不利于分析 uboot 的编译过程。可以通过设置变量“V=1“来实现完整的命令输出，这个在调试 uboot 的时候很有用，结果如图 31.3.3.2 所示：

```
  arm-linux-gnueabihf-ld.bfd   -pie  --gc-sections -Bstatic -Ttext 0x87800000 -o u-boot -T u-boot.lds arch/arm/c
pu/armv7/start.o --start-group  arch/arm/cpu/built-in.o  arch/arm/cpu/armv7/built-in.o  arch/arm/imx-common/buil
t-in.o  arch/arm/lib/built-in.o  board/freescale/common/built-in.o  board/freescale/mx6ull_alientek_emmc/built-i
n.o  cmd/built-in.o  common/built-in.o  disk/built-in.o  drivers/built-in.o  drivers/dma/built-in.o  drivers/gpi
o/built-in.o  drivers/i2c/built-in.o  drivers/mmc/built-in.o  drivers/mtd/built-in.o  drivers/mtd/onenand/built-
in.o  drivers/mtd/spi/built-in.o  drivers/net/built-in.o  drivers/net/phy/built-in.o  drivers/pci/built-in.o  dr
ivers/power/built-in.o  drivers/power/battery/built-in.o  drivers/power/fuel_gauge/built-in.o  drivers/power/mfd
/built-in.o  drivers/power/pmic/built-in.o  drivers/power/regulator/built-in.o  drivers/serial/built-in.o  drive
rs/spi/built-in.o  drivers/usb/dwc3/built-in.o  drivers/usb/emul/built-in.o  drivers/usb/eth/built-in.o  drivers
/usb/gadget/built-in.o  drivers/usb/gadget/udc/built-in.o  drivers/usb/host/built-in.o  drivers/usb/musb-new/bui
lt-in.o  drivers/usb/musb/built-in.o  drivers/usb/phy/built-in.o  drivers/usb/ulpi/built-in.o  fs/built-in.o  li
             net/built-in.o  test/built-in.o  test/dm/built-in.o --end-group arch/arm/lib/eabi_compat.o  -L /us
r/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/4.9.4 -lgcc -
Map u-boot.map
  arm-linux-gnueabihf-objcopy --gap-fill=0xff  -j .text -j .secure_text -j .rodata -j .hash -j .data -j .got -j
.got.plt -j .u_boot_list -j .rel.dyn -O binary  u-boot u-boot-nodtb.bin
  cp u-boot-nodtb.bin u-boot.bin
make -f ./scripts/Makefile.build obj=arch/arm/imx-common u-boot.imx
mkdir -p board/freescale/mx6ull_alientek_emmc/
  ./tools/mkimage -n board/freescale/mx6ull_alientek_emmc/imximage.cfg.cfgtmp -T imximage -e 0x87800000 -d u-boo
t.bin u-boot.imx
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:         DCD
Data Size:    425984 Bytes = 416.00 kB = 0.41 MB
Load Address: 877ff420
Entry Point:  87800000
  arm-linux-gnueabihf-objcopy --gap-fill=0xff  -j .text -j .secure_text -j .rodata -j .hash -j .data -j .got -j
.got.plt -j .u_boot_list -j .rel.dyn -O srec u-boot u-boot.srec
  arm-linux-gnueabihf-objdump -t u-boot > u-boot.sym
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 31.3.3.2 终端完整命令输出

顶层 Makefile 中控制命令输出的代码如下：

示例代码 31.3.3.1 顶层 Makefile 代码

```
73 ifeq ("$(origin V)", "command line")
74   KBUILD_VERBOSE = $(V)
75 endif
76 ifndef KBUILD_VERBOSE
77   KBUILD_VERBOSE = 0
78 endif
79
80 ifeq ($(KBUILD_VERBOSE),1)
81   quiet =
82   Q =
83 else
84   quiet=quiet_
85   Q = @
86 endif
```

上述代码中先使用 ifeq 来判断"$(origin V)"和"command line"是否相等。这里用到了 Makefile 中的函数 origin，origin 和其他的函数不一样，它不操作变量的值，origin 用于告诉你变量是哪来的，语法为：

```
$(origin <variable>)
```

variable 是变量名，o 函数的返回值就是变量来源，因此$(origin V)就是变量 V 的来源。如果变量 V 是在命令行定义的那么它的来源就是"command line"，这样"$(origin V)"和"command line"就相等了。当这两个相等的时候变量 KBUILD_VERBOSE 就等于 V 的值，比如在命令行中输入"V=1"的话那么 KBUILD_VERBOSE=1。如果没有在命令行输入 V 的话 KBUILD_VERBOSE=0。

第 80 行判断 KBUILD_VERBOSE 是否为 1，如果 KBUILD_VERBOSE 为 1 的话变量 quiet

和 Q 都为空，如果 KBUILD_VERBOSE=0 的话变量 quiet 为“quiet_“，变量 Q 为“@”,综上所述：

V=1 的话:

```
KBUILD_VERBOSE=1
quiet= 空 。
Q= 空。
```

V=0 或者命令行不定义 V 的话：

```
KBUILD_VERBOSE=0
quiet= quiet_。
Q= @。
```

Makefile 中会用到变量 quiet 和 Q 来控制编译的时候是否在终端输出完整的命令，在顶层 Makefile 中有很多如下所示的命令：

```
$(Q)$(MAKE) $(build)=tools
```

如果 V=0 的话上述命令展开就是“@ make $(build)=tools”，make 在执行的时候默认会在终端输出命令，但是在命令前面加上“@”就不会在终端输出命令了。当 V=1 的时候 Q 就为空，上述命令就是“make $(build)=tools”，因此在 make 执行的过程，命令会被完整的输出在终端上。

有些命令会有两个版本，比如：

```
quiet_cmd_sym ?= SYM      $@
cmd_sym ?= $(OBJDUMP) -t $< > $@
```

sym 命令分为“quiet_cmd_sym”和“cmd_sym”两个版本，这两个命令的功能都是一样的，区别在于 make 执行的时候输出的命令不同。quiet_cmd_xxx 命令输出信息少，也就是短命令，而 cmd_xxx 命令输出信息多，也就是完整的命令。

如果变量 quiet 为空的话，整个命令都会输出。

如果变量 quiet 为“quiet_”的话，仅输出短版本。

如果变量 quiet 为“silent_”的话，整个命令都不会输出。

31.3.4 静默输出

上一小节讲了，设置 V=0 或者在命令行中不定义 V 的话，编译 uboot 的时候终端中显示的短命令，但是还是会有命令输出，有时候我们在编译 uboot 的时候不需要输出命令，这个时候就可以使用 uboot 的静默输出功能。编译的时候使用“make -s”即可实现静默输出，顶层 Makefile 中相应的代码如下：

示例代码 31.3.4.1 顶层 Makefile 代码

```
88 # If the user is running make -s (silent mode), suppress echoing of
89 # commands
90
91 ifneq ($(filter 4.%,$(MAKE_VERSION)),)  # make-4
92 ifneq ($(filter %s ,$(firstword x$(MAKEFLAGS))),)
93   quiet=silent_
94 endif
95 else                    # make-3.8x
96 ifneq ($(filter s% -s%,$(MAKEFLAGS)),)
97   quiet=silent_
98 endif
```

```
99  endif
100
101 export quiet Q KBUILD_VERBOSE
```

第 91 行判断当前正在使用的编译器版本号是否为 4.x，判断$(filter 4.%,$(MAKE_VERSION))和“ ”(空)是否相等，如果不相等的话就成立，执行里面的语句。也就是说$(filter 4.%,$(MAKE_VERSION))不为空的话条件就成立，这里用到了 Makefile 中的 filter 函数，这是个过滤函数，函数格式如下：

```
$(filter <pattern...>,<text>)
```

filter 函数表示以 pattern 模式过滤 text 字符串中的单词，仅保留符合模式 pattern 的单词，可以有多个模式。函数返回值就是符合 pattern 的字符串。因此$(filter 4.%,$(MAKE_VERSION))的含义就是在字符串“MAKE_VERSION”中找出符合“4.%”的字符(%为通配符)，MAKE_VERSION 是编译器的版本号，我们当前使用的交叉编译器版本号为 4.9.4，所以肯定可以找出“4.%”。因此$(filter 4.%,$(MAKE_VERSION))不为空，条件成立，执行 92~94 行的语句。第 92 行也是一个判断语句，如果$(filter %s ,$(firstword x$(MAKEFLAGS)))不为空的话条件成立，变量 quiet 等于“silent_”。这里也用到了函数 filter，在$(firstword x$(MAKEFLAGS)))中过滤出符合“%s”的单词。到了函数 firstword，函数 firstword 是获取首单词，函数格式如下：

```
$(firstword <text>)
```

firstword 函数用于取出 text 字符串中的第一个单词，函数的返回值就是获取到的单词。当使用“make -s”编译的时候，“-s”会作为 MAKEFLAGS 变量的一部分传递给 Makefile。在顶层 Makfile 中添加如图 31.3.4.1 所示的代码：

```
85    Q = @
86 endif
87
88 # If the user is running make -s (silent mode), suppress echoing of
89 # commands
90
91 ifneq ($(filter 4.%,$(MAKE_VERSION)),)  # make-4
92 ifneq ($(filter %s ,$(firstword x$(MAKEFLAGS))),)
93   quiet=silent_
94 endif
95 else                    # make-3.8x
96 ifneq ($(filter s% -s%,$(MAKEFLAGS)),)
97   quiet=silent_
98 endif
99 endif
100
101 export quiet Q KBUILD_VERBOSE
102
103 mytest:
104     @echo 'firstword=' $(firstword x$(MAKEFLAGS))
105
106
107 # kbuild supports saving output files in a separate directory.
```


图 31.3.4.1 顶层 Makefile 添加代码

图 31.3.4.1 中的两行代码用于输出$(firstword x$(MAKEFLAGS))的结果，最后修改文件 mx6ull_alientek_emmc.sh，在里面加入“-s”选项，结果如图 31.3.4.2 所示：

```
zuozhongkai@ubuntu: ~/linux/IMX6ULL/uboot/alientek_uboot
1 #!/bin/bash
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_ddr512_emmc_defconfig
4 make -s ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j12
```


图 31.3.4.2 加入-s 选项

修改完成以后执行 mx6ull_alientek_emmc.sh，结果如图 31.3.4.3 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$ ./mx6ull_alientek_emmc.sh
  CLEAN   scripts/basic
  CLEAN   scripts/kconfig
  CLEAN   include/config include/generated
  CLEAN   .config include/autoconf.mk include/autoconf.mk.dep include/config.h
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#                                         第一个单词为 xrRs
firstword= xrRs
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 31.3.4.3 修改顶层 Makefile 后的执行结果

从图 31.3.4.3 可以看出第一个单词是“xrRs”，将$(filter %s ,$(firstword x$(MAKEFLAGS)))展开就是$(filter %s, xrRs)，而$(filter %s, xrRs)的返回值肯定不为空，条件成立，quiet=silent_。
第 101 行　　使用 export 导出变量 quiet、Q 和 KBUILD_VERBOSE。

31.3.5 设置编译结果输出目录

uboot 可以将编译出来的目标文件输出到单独的目录中，在 make 的时候使用“O”来指定输出目录，比如“make O=out”就是设置目标文件输出到 out 目录中。这么做是为了将源文件和编译产生的文件分开，当然也可以不指定 O 参数，不指定的话源文件和编译产生的文件都在同一个目录内，一般我们不指定 O 参数。顶层 Makefile 中相关的代码如下：

示例代码 31.3.5.1 顶层 Makefile 代码

```
103 # kbuild supports saving output files in a separate directory.
104 # To locate output files in a separate directory two syntaxes are
supported.
105 # In both cases the working directory must be the root of the
kernel src.
106 # 1) O=
107 # Use "make O=dir/to/store/output/files/"
108 #
109 # 2) Set KBUILD_OUTPUT
110 # Set the environment variable KBUILD_OUTPUT to point to the
directory
111 # where the output files shall be placed.
112 # export KBUILD_OUTPUT=dir/to/store/output/files/
113 # make
114 #
115 # The O= assignment takes precedence over the KBUILD_OUTPUT
environment
116 # variable.
117
118 # KBUILD_SRC is set on invocation of make in OBJ directory
119 # KBUILD_SRC is not intended to be used by the regular user (for
now)
120 ifeq ($(KBUILD_SRC),)
121
```

```
122 # OK, Make called in directory where kernel src resides
123 # Do we want to locate output files in a separate directory?
124 ifeq ("$(origin O)", "command line")
125   KBUILD_OUTPUT := $(O)
126 endif
127
128 # That's our default target when none is given on the command line
129 PHONY := _all
130 _all:
131
132 # Cancel implicit rules on top Makefile
133 $(CURDIR)/Makefile Makefile: ;
134
135 ifneq ($(KBUILD_OUTPUT),)
136 # Invoke a second make in the output directory, passing relevant
variables
137 # check that the output directory actually exists
138 saved-output := $(KBUILD_OUTPUT)
139 KBUILD_OUTPUT := $(shell mkdir -p $(KBUILD_OUTPUT) && cd
$(KBUILD_OUTPUT) \
140                                 && /bin/pwd)
......
155 endif # ifneq ($(KBUILD_OUTPUT),)
156 endif # ifeq ($(KBUILD_SRC),)
```

第 124 行判断“O”是否来自于命令行，如果来自命令行的话条件成立，KBUILD_OUTPUT 就为$(O)，因此变量 KBUILD_OUTPUT 就是输出目录。

第 135 行判断 KBUILD_OUTPUT 是否为空。

第 139 行调用 mkdir 命令，创建 KBUILD_OUTPUT 目录，并且将创建成功以后的绝对路径赋值给 KBUILD_OUTPUT。至此，通过 O 指定的输出目录就存在了。

31.3.6 代码检查

uboot 支持代码检查，使用命令“make C=1”使能代码检查，检查那些需要重新编译的文件。“make C=2”用于检查所有的源码文件，顶层 Makefile 中的代码如下：

示例代码 31.3.6.1 顶层 Makefile 代码

```
166 # Call a source code checker (by default, "sparse") as part of the
167 # C compilation.
168 #
169 # Use 'make C=1' to enable checking of only re-compiled files.
170 # Use 'make C=2' to enable checking of *all* source files,
regardless
171 # of whether they are re-compiled or not.
172 #
```

```
173 # See the file "Documentation/sparse.txt" for more details, including
174 # where to get the "sparse" utility.
175
176 ifeq ("$(origin C)", "command line")
177   KBUILD_CHECKSRC = $(C)
178 endif
179 ifndef KBUILD_CHECKSRC
180   KBUILD_CHECKSRC = 0
181 endif
```

第 176 行判断 C 是否来源于命令行，如果 C 来源于命令行，那就将 C 赋值给变量 KBUILD_CHECKSRC，如果命令行没有 C 的话 KBUILD_CHECKSRC 就为 0。

31.3.7 模块编译

在 uboot 中允许单独编译某个模块，使用命令“make M=dir”即可，旧语法“make SUBDIRS=dir”也是支持的。顶层 Makefile 中的代码如下：

示例代码 31.3.7.1 顶层 Makefile 代码

```
183 # Use make M=dir to specify directory of external module to build
184 # Old syntax make ... SUBDIRS=$PWD is still supported
185 # Setting the environment variable KBUILD_EXTMOD take precedence
186 ifdef SUBDIRS
187   KBUILD_EXTMOD ?= $(SUBDIRS)
188 endif
189
190 ifeq ("$(origin M)", "command line")
191   KBUILD_EXTMOD := $(M)
192 endif
193
194 # If building an external module we do not care about the all: rule
195 # but instead _all depend on modules
196 PHONY += all
197 ifeq ($(KBUILD_EXTMOD),)
198 _all: all
199 else
200 _all: modules
201 endif
202
203 ifeq ($(KBUILD_SRC),)
204         # building in the source tree
205         srctree := .
206 else
207         ifeq ($(KBUILD_SRC)/,$(dir $(CURDIR)))
208                 # building in a subdirectory of the source tree
```

```
209                 srctree := ..
210         else
211                 srctree := $(KBUILD_SRC)
212         endif
213 endif
214 objtree     := .
215 src     := $(srctree)
216 obj     := $(objtree)
217
218 VPATH       := $(srctree)$(if $(KBUILD_EXTMOD),:$(KBUILD_EXTMOD))
219
220 export srctree objtree VPATH
```

第 186 行判断是否定义了 SUBDIRS，如果定义了 SUBDIRS，变量 KBUILD_EXTMOD=SUBDIRS，这里是为了支持老语法“make SUBIDRS=dir”

第 190 行判断是否在命令行定义了 M，如果定义了的话 KBUILD_EXTMOD=$(M)。

第 197 行判断 KBUILD_EXTMOD 时为空，如果为空的话目标_all 依赖 all，因此要先编译出 all。否则的话默认目标_all 依赖 modules，要先编译出 modules，也就是编译模块。一般情况下我们不会在 uboot 中编译模块，所以此处会编译 all 这个目标。

第 203 行判断 KBUILD_SRC 是否为空，如果为空的话就设置变量 srctree 为当前目录，即 srctree 为“.”，一般不设置 KBUILD_SRC。

第 214 行设置变量 objtree 为当前目录。

第 215 和 216 行分别设置变量 src 和 obj，都为当前目录。

第 218 行设置 VPATH。

第 220 行导出变量 scrtree、objtree 和 VPATH。

31.3.8 获取主机架构和系统

接下来顶层 Makefile 会获取主机架构和系统，也就是我们电脑的架构和系统，代码如下：

示例代码 31.3.8.1 顶层 Makefile 代码

```
227 HOSTARCH := $(shell uname -m | \
228     sed -e s/i.86/x86/ \
229         -e s/sun4u/sparc64/ \
230         -e s/arm.*/arm/ \
231         -e s/sa110/arm/ \
232         -e s/ppc64/powerpc/ \
233         -e s/ppc/powerpc/ \
234         -e s/macppc/powerpc/\
235         -e s/sh.*/sh/)
236
237 HOSTOS := $(shell uname -s | tr '[:upper:]' '[:lower:]' | \
238         sed -e 's/\(cygwin\).*/cygwin/')
239
240 export  HOSTARCH HOSTOS
```

第 227 行定义了一个变量 HOSTARCH，用于保存主机架构，这里调用 shell 命令“uname -

m”获取架构名称，结果如图 31.3.8.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ uname -m
x86_64
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 31.3.8.1 uname -m 命令

从图 31.3.8.1 可以看出当前电脑主机架构为“x86_64”，shell 中的“|”表示管道，意思是将左边的输出作为右边的输入，sed -e 是替换命令，“sed -e s/i.86/x86/”表示将管道输入的字符串中的“i.86”替换为“x86”，其他的“sed -s”命令同理。对于我的电脑而言，HOSTARCH=x86_64。

第 237 行定义了变量 HOSTOS，此变量用于保存主机 OS 的值，先使用 shell 命令“name -s”来获取主机 OS，结果如图 31.3.8.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$ uname -s
Linux
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 31.3.8.2 uname -s 命令

从图 31.3.8.2 可以看出此时的主机 OS 为“Linux”，使用管道将“Linux”作为后面“tr '[:upper:]' '[:lower:]'”的输入，“tr '[:upper:]' '[:lower:]'”表示将所有的大写字母替换为小写字母，因此得到“linux”。最后同样使用管道，将“linux”作为“sed -e 's/\(cygwin\).*/cygwin/'”的输入，用于将 cygwin.*替换为 cygwin。因此，HOSTOS=linux。

第 240 行导出 HOSTARCH=x86_64，HOSTOS=linux。

31.3.9 设置目标架构、交叉编译器和配置文件

编译 uboot 的时候需要设置目标板架构和交叉编译器，“make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-”就是用于设置 ARCH 和 CROSS_COMPILE，在顶层 Makefile 中代码如下：

示例代码 31.3.9.1 顶层 Makefile 代码

```
244 # set default to nothing for native builds
245 ifeq ($(HOSTARCH),$(ARCH))
246 CROSS_COMPILE ?=
247 endif
248
249 KCONFIG_CONFIG  ?= .config
250 export KCONFIG_CONFIG
```

第 245 行判断 HOSTARCH 和 ARCH 这两个变量是否相等，主机架构(变量 HOSTARCH)是 x86_64，而我们编译的是 ARM 版本 uboot，肯定不相等，所以 CROS_COMPILE= arm-linux-gnueabihf-。从示例代码 31.3.9.1 可以看出，每次编译 uboot 的时候都要在 make 命令后面设置 ARCH 和 CROS_COMPILE，使用起来很麻烦，可以直接修改顶层 Makefile，在里面加入 ARCH 和 CROSS_COMPILE 的定义，如图 31.3.9.1 所示：

```
244 # set default to nothing for native builds
245 ifeq ($(HOSTARCH),$(ARCH))
246 CROSS_COMPILE ?=
247 endif
248
249 ARCH ?= arm
250 CROSS_COMPILE ?= arm-linux-gnueabihf-
251
252 KCONFIG_CONFIG  ?= .config
253 export KCONFIG_CONFIG
```

图 31.3.9.1 定义 ARCH 和 CROSS_COMPILE

按照图 31.3.9.1 所示，直接在顶层 Makefile 里面定义 ARCH 和 CROSS_COMPILE，这样就不用每次编译的时候都要在 make 命令后面定义 ARCH 和 CROSS_COMPILE。

第 249 行定义变量 KCONFIG_CONFIG，uboot 是可以配置的，这里设置配置文件为.config，.config 默认是没有的，需要使用命令“make xxx_defconfig”对 uboot 进行配置，配置完成以后就会在 uboot 根目录下生成.config。默认情况下.config 和 xxx_defconfig 内容是一样的，因为.config 就是从 xxx_defconfig 复制过来的。如果后续自行调整了 uboot 的一些配置参数，那么这些新的配置参数就添加到了.config 中，而不是 xxx_defconfig。相当于 xxx_defconfig 只是一些初始配置，而.config 里面的才是实时有效的配置。

31.3.10 调用 scripts/Kbuild.include

主 Makefile 会调用文件 scripts/Kbuild.include 这个文件，顶层 Makefile 中代码如下：

示例代码 31.3.10.1 顶层 Makefile 代码

```
327 # We need some generic definitions (do not try to remake the file).
328 scripts/Kbuild.include: ;
329 include scripts/Kbuild.include
```

示例代码 31.3.10.1 中使用“include”包含了文件 scripts/Kbuild.include，此文件里面定义了很多变量，如图 31.3.10.1 所示：

```
1  ####
2  # kbuild: Generic definitions
3
4  # Convenient variables
5  comma   := ,
6  quote   := "
7  squote  := '
8  empty   :=
9  space   := $(empty) $(empty)
0
1  ###
2  # Name of target with a '.' as filename prefix. foo/bar.o => foo/.bar.o
3  dot-target = $(dir $@).$(notdir $@)
4
5  ###
6  # The temporary file to save gcc -MD generated dependencies must not
7  # contain a comma
8  depfile = $(subst $(comma),_,$(dot-target).d)
9
```

图 31.3.10.1 Kbuild.include 文件

在 uboot 的编译过程中会用到 scripts/Kbuild.include 中的这些变量，后面用到的时候再分析。

31.3.11 交叉编译工具变量设置

上面我们只是设置了 CROSS_COMPILE 的名字，但是交叉编译器其他的工具还没有设置，顶层 Makefile 中相关代码如下：

示例代码 31.3.11.1 顶层 Makefile 代码

```
331 # Make variables (CC, etc...)
332
333 AS          = $(CROSS_COMPILE)as
334 # Always use GNU ld
335 ifneq ($(shell $(CROSS_COMPILE)ld.bfd -v 2> /dev/null),)
```

```
336 LD          = $(CROSS_COMPILE)ld.bfd
337 else
338 LD          = $(CROSS_COMPILE)ld
339 endif
340 CC          = $(CROSS_COMPILE)gcc
341 CPP         = $(CC) -E
342 AR          = $(CROSS_COMPILE)ar
343 NM          = $(CROSS_COMPILE)nm
344 LDR         = $(CROSS_COMPILE)ldr
345 STRIP       = $(CROSS_COMPILE)strip
346 OBJCOPY     = $(CROSS_COMPILE)objcopy
347 OBJDUMP     = $(CROSS_COMPILE)objdump
```

31.3.12 导出其他变量

接下来在顶层 Makefile 会导出很多变量，代码如下：

示例代码 31.3.12.1 顶层 Makefile 代码

```
368 export VERSION PATCHLEVEL SUBLEVEL UBOOTRELEASE UBOOTVERSION
369 export ARCH CPU BOARD VENDOR SOC CPUDIR BOARDDIR
370 export CONFIG_SHELL HOSTCC HOSTCFLAGS HOSTLDFLAGS CROSS_COMPILE AS
LD CC
371 export CPP AR NM LDR STRIP OBJCOPY OBJDUMP
372 export MAKE AWK PERL PYTHON
373 export HOSTCXX HOSTCXXFLAGS DTC CHECK CHECKFLAGS
374
375 export KBUILD_CPPFLAGS NOSTDINC_FLAGS UBOOTINCLUDE OBJCOPYFLAGS
LDFLAGS
376 export KBUILD_CFLAGS KBUILD_AFLAGS
```

这些变量中大部分都已经在前面定义了，我们重点来看一下下面这几个变量：

```
ARCH CPU BOARD VENDOR SOC CPUDIR BOARDDIR
```

这 7 个变量在顶层 Makefile 是找不到的，说明这 7 个变量是在其他文件里面定义的，先来看一下这 7 个变量都是什么内容，在顶层 Makefile 中输入如图 31.3.12.1 所示的内容：

```
372 export VERSION PATCHLEVEL SUBLEVEL UBOOTRELEASE UBOOTVERSION
373 export ARCH CPU BOARD VENDOR SOC CPUDIR BOARDDIR
374 export CONFIG_SHELL HOSTCC HOSTCFLAGS HOSTLDFLAGS CROSS_COMPILE AS LD CC
375 export CPP AR NM LDR STRIP OBJCOPY OBJDUMP
376 export MAKE AWK PERL PYTHON
377 export HOSTCXX HOSTCXXFLAGS DTC CHECK CHECKFLAGS
378
379 export KBUILD_CPPFLAGS NOSTDINC_FLAGS UBOOTINCLUDE OBJCOPYFLAGS LDFLAGS
380 export KBUILD_CFLAGS KBUILD_AFLAGS
381
382 mytest:
383     @echo 'ARCH=' $(ARCH)
384     @echo 'CPU=' $(CPU)
385     @echo 'BOARD=' $(BOARD)
386     @echo 'VENDOR=' $(VENDOR)
387     @echo 'SOC=' $(SOC)
388     @echo 'CPUDIR=' $(CPUDIR)
389     @echo 'BOARDDIR=' $(BOARDDIR)
390
```

输入这几行代码

图 31.3.12.1 输出变量值

修改好顶层 Makefile 以后执行如下命令：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-    mytest
```

结果如图 31.3.12.2 所示：

```
ARCH= arm
CPU= armv7
BOARD= mx6ullevk
VENDOR= freescale
SOC= mx6
CPUDIR= arch/arm/cpu/armv7
BOARDDIR= freescale/mx6ullevk
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 31.3.12.2 变量结果

从图 31.3.12.2 可以看到这 7 个变量的值，这 7 个变量是从哪里来的呢？在 uboot 根目录下有个文件叫做 config.mk，这 7 个变量就是在 config.mk 里面定义的，打开 config.mk 内容如下：

示例代码 31.3.12.2 config.mk 代码

```
1  #
2  # (C) Copyright 2000-2013
3  # Wolfgang Denk, DENX Software Engineering, wd@denx.de.
4  #
5  # SPDX-License-Identifier:   GPL-2.0+
6  #
7  
#########################################################################
##
8
9  # This file is included from ./Makefile and spl/Makefile.
10 # Clean the state to avoid the same flags added twice.
11 #
12 # (Tegra needs different flags for SPL.
13 #  That's the reason why this file must be included from spl/Makefile
too.
14 #  If we did not have Tegra SoCs, build system would be much
simpler...)
15 PLATFORM_RELFLAGS :=
16 PLATFORM_CPPFLAGS :=
17 PLATFORM_LDFLAGS :=
18 LDFLAGS :=
19 LDFLAGS_FINAL :=
20 OBJCOPYFLAGS :=
21 # clear VENDOR for tcsh
22 VENDOR :=
23 
#########################################################################
##
24
25 ARCH := $(CONFIG_SYS_ARCH:"%"=%)
26 CPU := $(CONFIG_SYS_CPU:"%"=%)
```

```
27 ifdef CONFIG_SPL_BUILD
28 ifdef CONFIG_TEGRA
29 CPU := arm720t
30 endif
31 endif
32 BOARD := $(CONFIG_SYS_BOARD:"%"=%)
33 ifneq ($(CONFIG_SYS_VENDOR),)
34 VENDOR := $(CONFIG_SYS_VENDOR:"%"=%)
35 endif
36 ifneq ($(CONFIG_SYS_SOC),)
37 SOC := $(CONFIG_SYS_SOC:"%"=%)
38 endif
39
40 # Some architecture config.mk files need to know what CPUDIR is set
to,
41 # so calculate CPUDIR before including ARCH/SOC/CPU config.mk files.
42 # Check if arch/$ARCH/cpu/$CPU exists, otherwise assume
arch/$ARCH/cpu contains
43 # CPU-specific code.
44 CPUDIR=arch/$(ARCH)/cpu$(if $(CPU),/$(CPU),)
45
46 sinclude $(srctree)/arch/$(ARCH)/config.mk
47 sinclude $(srctree)/$(CPUDIR)/config.mk
48
49 ifdef   SOC
50 sinclude $(srctree)/$(CPUDIR)/$(SOC)/config.mk
51 endif
52 ifneq ($(BOARD),)
53 ifdef   VENDOR
54 BOARDDIR = $(VENDOR)/$(BOARD)
55 else
56 BOARDDIR = $(BOARD)
57 endif
58 endif
59 ifdef   BOARD
60 sinclude $(srctree)/board/$(BOARDDIR)/config.mk  # include board
specific rules
61 endif
62
63 ifdef FTRACE
64 PLATFORM_CPPFLAGS += -finstrument-functions -DFTRACE
65 endif
66
```

```
67 # Allow use of stdint.h if available
68 ifneq ($(USE_STDINT),)
69 PLATFORM_CPPFLAGS += -DCONFIG_USE_STDINT
70 endif
71
72
#########################################################################
##
73
74 RELFLAGS := $(PLATFORM_RELFLAGS)
75
76 PLATFORM_CPPFLAGS += $(RELFLAGS)
77 PLATFORM_CPPFLAGS += -pipe
78
79 LDFLAGS += $(PLATFORM_LDFLAGS)
80 LDFLAGS_FINAL += -Bstatic
81
82 export PLATFORM_CPPFLAGS
83 export RELFLAGS
84 export LDFLAGS_FINAL
```

第 25 行定义变量 ARCH，值为$(CONFIG_SYS_ARCH:"%"=%)，也就是提取 CONFIG_SYS_ARCH 里面双引号“”之间的内容。比如 CONFIG_SYS_ARCH=“arm”的话，ARCH=arm。

第 26 行定义变量 CPU，值为$(CONFIG_SYS_CPU:"%"=%)。

第 32 行定义变量 BOARD，值为(CONFIG_SYS_BOARD:"%"=%)。

第 34 行定义变量 VENDOR，值为$(CONFIG_SYS_VENDOR:"%"=%)。

第 37 行定义变量 SOC，值为$(CONFIG_SYS_SOC:"%"=%)。

第 44 行定义变量 CPUDIR，值为 arch/$(ARCH)/cpu$(if $(CPU),/$(CPU),)。

第 46 行 sinclude 和 include 的功能类似，在 Makefile 中都是读取指定文件内容，这里读取文件$(srctree)/arch/$(ARCH)/config.mk 的内容。sinclude 读取的文件如果不存在的话不会报错。

第 47 行读取文件$(srctree)/$(CPUDIR)/config.mk 的内容。

第 50 行读取文件$(srctree)/$(CPUDIR)/$(SOC)/config.mk 的内容。

第 54 行定义变量 BOARDDIR，如果定义了 VENDOR 那么 BOARDDIR=$(VENDOR)/$(BOARD)，否则的 BOARDDIR=$(BOARD)。

第 60 行读取文件$(srctree)/board/$(BOARDDIR)/config.mk。

接下来需要找到 CONFIG_SYS_ARCH、CONFIG_SYS_CPU、CONFIG_SYS_BOARD、CONFIG_SYS_VENDOR 和 CONFIG_SYS_SOC 这 5 个变量的值。这 5 个变量在 uboot 根目录下的.config 文件中有定义，定义如下：

示例代码 31.3.12.3 .config 文件代码

```
23 CONFIG_SYS_ARCH="arm"
24 CONFIG_SYS_CPU="armv7"
25 CONFIG_SYS_SOC="mx6"
26 CONFIG_SYS_VENDOR="freescale"
```

```
27 CONFIG_SYS_BOARD="mx6ullevk "
28 CONFIG_SYS_CONFIG_NAME="mx6ullevk"
```

根据示例代码 31.3.12.3 可知：

```
ARCH = arm
CPU = armv7
BOARD = mx6ullevk
VENDOR = freescale
SOC = mx6
CPUDIR = arch/arm/cpu/armv7
BOARDDIR = freescale/mx6ullevk
```

在 config.mk 中读取的文件有：

```
arch/arm/config.mk
arch/arm/cpu/armv7/config.mk
arch/arm/cpu/armv7/mx6/config.mk (此文件不存在)
board/ freescale/mx6ullevk/config.mk (此文件不存在)
```

31.3.13 make xxx_defconfig 过程

在编译 uboot 之前要使用“make xxx_defconfig”命令来配置 uboot，那么这个配置过程是如何运行的呢？在顶层 Makefile 中有如下代码：

示例代码 31.3.13.1 顶层 Makefile 代码段

```
414 # To make sure we do not include .config for any of the *config
targets
415 # catch them early, and hand them over to scripts/kconfig/Makefile
416 # It is allowed to specify more targets when calling make,
including
417 # mixing *config targets and build targets.
418 # For example 'make oldconfig all'.
419 # Detect when mixed targets is specified, and make a second
invocation
420 # of make so .config is not included in this case either (for
*config).
421
422 version_h := include/generated/version_autogenerated.h
423 timestamp_h := include/generated/timestamp_autogenerated.h
424
425 no-dot-config-targets := clean clobber mrproper distclean \
426            help %docs check% coccicheck \
427            ubootversion backup
428
429 config-targets := 0
430 mixed-targets  := 0
431 dot-config     := 1
432
```

```
433 ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
434     ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
435         dot-config := 0
436     endif
437 endif
438
439 ifeq ($(KBUILD_EXTMOD),)
440         ifneq ($(filter config %config,$(MAKECMDGOALS)),)
441                 config-targets := 1
442                 ifneq ($(words $(MAKECMDGOALS)),1)
443                         mixed-targets := 1
444                 endif
445         endif
446 endif
447
448 ifeq ($(mixed-targets),1)
449 # ================================================================
450 # We're called with mixed targets (*config and build targets).
451 # Handle them one by one.
452
453 PHONY += $(MAKECMDGOALS) __build_one_by_one
454
455 $(filter-out __build_one_by_one, $(MAKECMDGOALS)):
__build_one_by_one
456     @:
457
458 __build_one_by_one:
459     $(Q)set -e; \
460     for i in $(MAKECMDGOALS); do \
461         $(MAKE) -f $(srctree)/Makefile $$i; \
462     done
463
464 else
465 ifeq ($(config-targets),1)
466 # ================================================================
467 # *config targets only - make sure prerequisites are updated, and
descend
468 # in scripts/kconfig to make the *config target
469
470 KBUILD_DEFCONFIG := sandbox_defconfig
471 export KBUILD_DEFCONFIG KBUILD_KCONFIG
472
473 config: scripts_basic outputmakefile FORCE
```

```
474     $(Q)$(MAKE) $(build)=scripts/kconfig $@
475
476 %config: scripts_basic outputmakefile FORCE
477     $(Q)$(MAKE) $(build)=scripts/kconfig $@
478
479 else
480 #==================================================================
481 # Build targets only - this includes vmlinux, arch specific
targets, clean
482 # targets and others. In general all targets except *config
targets.
483
484 ifeq ($(dot-config),1)
485 # Read in config
486 -include include/config/auto.conf
......
```

第 422 行定义了变量 version_h，这变量保存版本号文件，此文件是自动生成的。文件 include/generated/version_autogenerated.h 内容如图 31.3.13.1 所示：

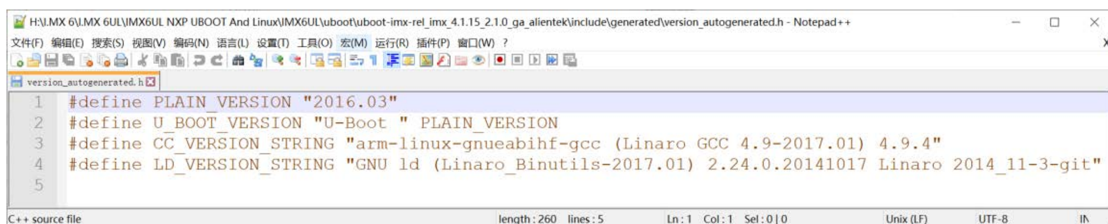


图 31.3.13.1 版本号文件

第 423 行定义了变量 timestamp_h，此变量保存时间戳文件，此文件也是自动生成的。文件 include/generated/timestamp_autogenerated.h 内容如图 31.3.13.2 所示：

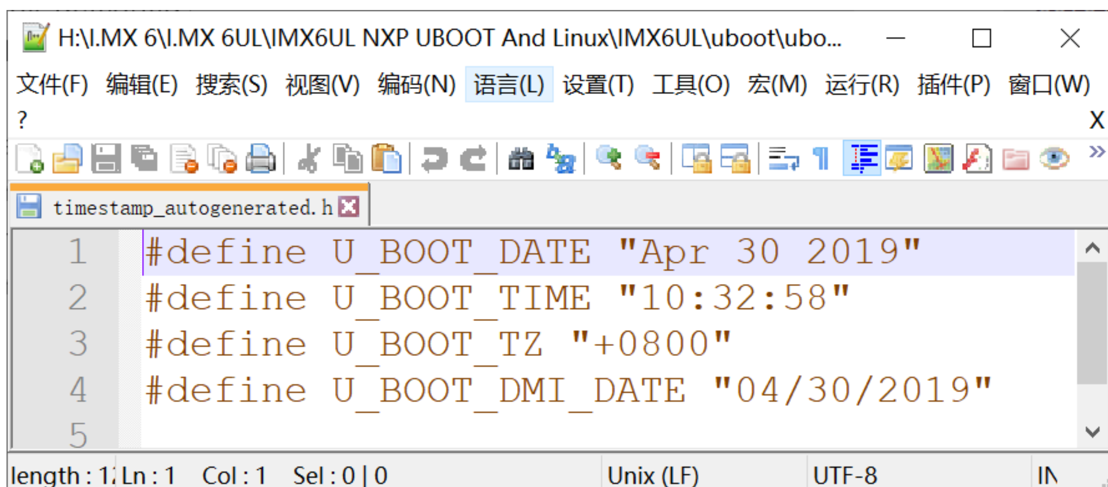


图 31.3.13.2 时间戳文件

第 425 行定义了变量 no-dot-config-targets。

第 429 行定义了变量 config-targets，初始值为 0。

第 430 行定义了变量 mixed-targets，初始值为 0。

第 431 行定义了变量 dot-config，初始值为 1。

第 433 行将 MAKECMDGOALS 中不符合 no-dot-config-targets 的部分过滤掉，剩下的如果不为空的话条件就成立。MAKECMDGOALS 是 make 的一个环境变量，这个变量会保存你所指定的终极目标列表，比如执行"make mx6ull_alientek_emmc_defconfig"，那么 MAKECMDGOALS 就为 mx6ull_alientek_emmc_defconfig。很明显过滤后为空，所以条件不成立，变量 dot-config 依旧为 1。

第 439 行判断 KBUILD_EXTMOD 是否为空，如果 KBUILD_EXTMOD 为空的话条件成立，经过前面的分析，我们知道 KBUILD_EXTMOD 为空，所以条件成立。

第 440 行将 MAKECMDGOALS 中不符合"config"和"%config"的部分过滤掉，如果剩下的部分不为空条件就成立，很明显此处条件成立，变量 config-targets=1。

第 442 行统计 MAKECMDGOALS 中的单词个数，如果不为 1 的话条件成立。此处调用 Makefile 中的 words 函数来统计单词个数，words 函数格式如下：

```
$(words <text>)
```

很明显，MAKECMDGOALS 的单词个数是 1 个，所以条件不成立，mixed-targets 继续为 0。综上所述，这些变量值如下：

```
config-targets = 1
mixed-targets = 0
dot-config = 1
```

第 448 行如果变量 mixed-targets 为 1 的话条件成立，很明显，条件不成立。

第 465 行如果变量 config-targets 为 1 的话条件成立，很明显，条件成立，执行这个分支。

第 473 行，没有目标与之匹配，所以不执行。

第 476 行，有目标与之匹配，当输入"make xxx_defconfig"的时候就会匹配到%config 目标，目标"%config"依赖于 scripts_basic、outputmakefile 和 FORCE。FORCE 在顶层 Makefile 的 1610 行有如下定义：

示例代码 31.3.13.2 顶层 Makefile 代码段

```
1610 PHONY += FORCE
1611 FORCE:
```

可以看出 FORCE 是没有规则和依赖的，所以每次都会重新生成 FORCE。当 FORCE 作为其他目标的依赖时，由于 FORCE 总是被更新过的，因此依赖所在的规则总是会执行的。

依赖 scripts_basic 和 outputmakefile 在顶层 Makefile 中的内容如下：

示例代码 31.3.13.3 顶层 Makefile 代码段

```
394 # Basic helpers built in scripts/
395 PHONY += scripts_basic
396 scripts_basic:
397     $(Q)$(MAKE) $(build)=scripts/basic
398     $(Q)rm -f .tmp_quiet_recordmcount
399 
400 # To avoid any implicit rule to kick in, define an empty command.
401 scripts/basic/%: scripts_basic ;
402 
403 PHONY += outputmakefile
404 # outputmakefile generates a Makefile in the output directory, if
405 # using a separate output directory. This allows convenient use of
```

```
406 # make in the output directory.
407 outputmakefile:
408 ifneq ($(KBUILD_SRC),)
409     $(Q)ln -fsn $(srctree) source
410     $(Q)$(CONFIG_SHELL) $(srctree)/scripts/mkmakefile \
411         $(srctree) $(objtree) $(VERSION) $(PATCHLEVEL)
412 endif
```

第 408 行，判断 KBUILD_SRC 是否为空，只有变量 KBUILD_SRC 不为空的时候 outputmakefile 才有意义，经过我们前面的分析 KBUILD_SRC 为空，所以 outputmakefile 无效。只有 scripts_basic 是有效的。

第 396~398 行是 scripts_basic 的规则，其对应的命令用到了变量 Q、MAKE 和 build，其中：

```
Q=@或为空
MAKE=make
```

变量 build 是在 scripts/Kbuild.include 文件中有定义，定义如下：

示例代码 31.3.13.3 Kbuild.include 代码段

```
177 ###
178 # Shorthand for $(Q)$(MAKE) -f scripts/Makefile.build obj=
179 # Usage:
180 # $(Q)$(MAKE) $(build)=dir
181 build := -f $(srctree)/scripts/Makefile.build obj
```

从示例代码 31.3.13.3 可以看出 build=-f $(srctree)/scripts/Makefile.build obj，经过前面的分析可知，变量 srctree 为”.”，因此：

```
build=-f ./scripts/Makefile.build obj
```

scripts_basic 展开以后如下：

```
scripts_basic:
    @make -f ./scripts/Makefile.build obj=scripts/basic     //也可以没有@，视配置而定
    @rm -f . tmp_quiet_recordmcount                          //也可以没有@
```

scripts_basic 会调用文件./scripts/Makefile.build，这个我们后面在分析。

接着回到示例代码 31.3.13.1 中的%config 处，内容如下：

```
%config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

将命令展开就是：

```
@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig
```

同样也跟文件./scripts/Makefile.build 有关，我们后面再分析此文件。使用如下命令配置 uboot，并观察其配置过程：

```
make mx6ull_14x14_ddr512_emmc_defconfig V=1
```

配置过程如图 31.3.13.1 所示。

```
make -f ./scripts/Makefile.build obj=scripts/basic
rm -f .tmp_quiet_recordmcount
make -f ./scripts/Makefile.build obj=scripts/kconfig mx6ull_14x14_ddr512_emmc_defconfig
scripts/kconfig/conf  --defconfig=arch/../configs/mx6ull_14x14_ddr512_emmc_defconfig Kconfig
#
# configuration written to .config
#
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

目标scripts_basic对应的命令
目标%config对应的命令

图 31.3.13.1 uboot 配置过程

从图 31.3.13.1 可以看出，我们的分析是正确的，接下来就要结合下面两行命令重点分析一下文件 scripts/Makefile.build。

①、scripts_basic 目标对应的命令

```
@make -f ./scripts/Makefile.build obj=scripts/basic
```

②、%config 目标对应的命令

```
@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig
```

31.3.14 Makefile.build 脚本分析

从上一小节可知，“make xxx_defconfig“配置 uboot 的时候如下两行命令会执行脚本 scripts/Makefile.build：

```
@make -f ./scripts/Makefile.build obj=scripts/basic
@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig
```

依次来分析一下：

1、scripts_basic 目标对应的命令

scripts_basic 目标对应的命令为：@make -f ./scripts/Makefile.build obj=scripts/basic。打开文件 scripts/Makefile.build，有如下代码：

示例代码 31.3.14.1 Makefile.build 代码段

```
8  # Modified for U-Boot
9  prefix := tpl
10 src := $(patsubst $(prefix)/%,%,$(obj))
11 ifeq ($(obj),$(src))
12 prefix := spl
13 src := $(patsubst $(prefix)/%,%,$(obj))
14 ifeq ($(obj),$(src))
15 prefix := .
16 endif
17 endif
```

第 9 行定义了变量 prefix 值为 tpl。

第 10 行定义了变量 src，这里用到了函数 patsubst，此行代码展开后为：

```
$(patsubst tpl/%,%, scripts/basic)
```

patsubst 是替换函数，格式如下：

```
$(patsubst <pattern>,<replacement>,<text>)
```

此函数用于在 text 中查找符合 pattern 的部分，如果匹配的话就用 replacement 替换掉。pattenr 是可以包含通配符“%”，如果 replacement 中也包含通配符“%”，那么 replacement 中的这个“%”将是 pattern 中的那个“%”所代表的字符串。函数的返回值为替换后的字符串。因此，第 10 行就是在“scripts/basic”中查找符合“tpl/%”的部分，然后将“tpl/”取消掉，但是“scripts/basic”没有“tpl/”，所以 src= scripts/basic。

第 11 行判断变量 obj 和 src 是否相等，相等的话条件成立，很明显，此处条件成立。

第 12 行和第 9 行一样，只是这里处理的是“spl”，“scripts/basic”里面也没有“spl/”，所以 src 继续为 scripts/basic。

第 15 行因为变量 obj 和 src 相等，所以 prefix=.。

继续分析 scripts/Makefile.build，有如下代码：

示例代码 31.3.14.2 Makefile.build 代码段

```
56 # The filename Kbuild has precedence over Makefile
57 kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
58 kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-
dir)/Kbuild,$(kbuild-dir)/Makefile)
59 include $(kbuild-file)
```

将 kbuild-dir 展开后为：

```
$(if $(filter /%, scripts/basic),    scripts/basic, ./scripts/basic),
```

因为没有以“/”为开头的单词，所以$(filter /%, scripts/basic)的结果为空，kbuild-dir=./scripts/basic。

将 kbuild-file 展开后为：

```
$(if $(wildcard ./scripts/basic/Kbuild), ./scripts/basic/Kbuild, ./scripts/basic/Makefile)
```

因为 scrpts/basic 目录中没有 Kbuild 这个文件，所以 kbuild-file= ./scripts/basic/Makefile。最后将 59 行展开，即：

```
include ./scripts/basic/Makefile
```

也就是读取 scripts/basic 下面的 Makefile 文件。

继续分析 scripts/Makefile.build，如下代码：

示例代码 31.3.14.3 Makefile.build 代码段

```
116 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target)
$(extra-y)) \
117      $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
118      $(subdir-ym) $(always)
119     @:
```

__build 是默认目标，因为命令“@make -f ./scripts/Makefile.build obj=scripts/basic”没有指定目标，所以会使用到默认目标：__build。在顶层 Makefile 中，KBUILD_BUILTIN 为 1，KBUILD_MODULES 为 0，因此展开后目标__build 为：

```
__build:$(builtin-target) $(lib-target) $(extra-y)) $(subdir-ym) $(always)
    @:
```

可以看出目标__build 有 5 个依赖：builtin-target、lib-target、extra-y、subdir-ym 和 always。这 5 个依赖的具体内容我们就不通过源码来分析了，直接在 scripts/Makefile.build 中输入图 31.3.14.1 所示内容，将这 5 个变量的值打印出来：

```
117 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target) $(extra-y)) \
118      $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
119      $(subdir-ym) $(always)
120     @:
121     @echo builtin-target = $(builtin-target)
122     @echo lib-target = $(lib-target)
123     @echo extra-y = $(extra-y)
124     @echo subdir-ym = $(subdir-ym)
125     @echo always = $(always)
126
```

输出5个依赖的值

图 31.3.14.1 输出变量

执行如下命令：

```
make mx6ull_14x14_ddr512_emmc_defconfig V=1
```

结果如图 31.3.14.2 所示：

```
make -f ./scripts/Makefile.build obj=scripts/basic
builtin-target =
lib-target =
extra-y =
subdir-ym =
always = scripts/basic/fixdep
rm -f .tmp_quiet_recordmcount
make -f ./scripts/Makefile.build obj=scripts/kconfig mx6ull_alientek_emmc_defconfig
scripts/kconfig/conf  --defconfig=arch/../configs/mx6ull_alientek_emmc_defconfig Kconfig
#
# configuration written to .config
#
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 31.3.14.2 输出结果

从上图可以看出，只有 always 有效，因此__build 最终为：

```
__build: scripts/basic/fixdep
    @:
```

__build 依赖于 scripts/basic/fixdep，所以要先 scripts/basic/fixdep.c 编译，生成 fixdep，前面已经读取了 scripts/basic/Makefile 文件。

综上所述，scripts_basic 目标的作用就是编译出 scripts/basic/fixdep 这个软件。

2、%config 目标对应的命令

%config 目标对应的命令为：@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig，各个变量值如下：

```
src= scripts/kconfig
kbuild-dir = ./scripts/kconfig
kbuild-file = ./scripts/kconfig/Makefile
include ./scripts/kconfig/Makefile
```

可以看出，Makefilke.build 会读取 scripts/kconfig/Makefile 中的内容，此文件有如下所示内容：

示例代码 31.3.14.4 scripts/kconfig/Makefile 代码段

```
113 %_defconfig: $(obj)/conf
114     $(Q)$< $(silent) --defconfig=arch/$(SRCARCH)/configs/$@
$(Kconfig)
115
116 # Added for U-Boot (backward compatibility)
117 %_config: %_defconfig
118     @:
```

目标%_defconfig 刚好和我们输入的 xxx_defconfig 匹配，所以会执行这条规则。依赖为$(obj)/conf，展开后就是 scripts/kconfig/conf。接下来就是检查并生成依赖 scripts/kconfig/conf。conf 是主机软件，到这里我们就打住，不要纠结 conf 是怎么编译出来的，否则就越陷越深，太绕了，像 conf 这种主机所使用的工具类软件我们一般不关心它是如何编译产生的。如果一定要看是 conf 是怎么生成的，可以输入如下命令重新配置 uboot，在重新配置 uboot 的过程中就会输出 conf 编译信息。

```
make distclean
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_ddr512_emmc_
defconfig   V=1
```

结果如图 31.3.14.3 所示：

```
  cc  -o scripts/kconfig/conf scripts/kconfig/conf.o scripts/kconfig/zconf.tab.o
scripts/kconfig/conf  --defconfig=arch/../configs/mx6ull_14x14_ddr512_emmc_defconfig Kconfig
#
# configuration written to .config                                 编译生成conf
#
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/alientek_uboot$
```

图 31.3.14.3 编译过程

得到 scripts/kconfig/conf 以后就要执行目标%_defconfig 的命令：

(Q)< $(silent) --defconfig=arch/$(SRCARCH)/configs/$@ $(Kconfig)

相关的变量值如下：

silent=-s 或为空
SRCARCH=..
Kconfig=Kconfig

将其展开就是：

@ scripts/kconfig/conf --defconfig=arch/../configs/xxx_defconfig Kconfig

上述命令用到了 xxx_defconfig 文件，比如 mx6ull_alientek_emmc_defconfig。这里会将 mx6ull_alientek_emmc_defconfig 中的配置输出到.config 文件中，最终生成 uboot 根目录下的.config 文件。

这个就是命令 make xxx_defconfig 执行流程，总结一下如图 31.3.14.4 所示：

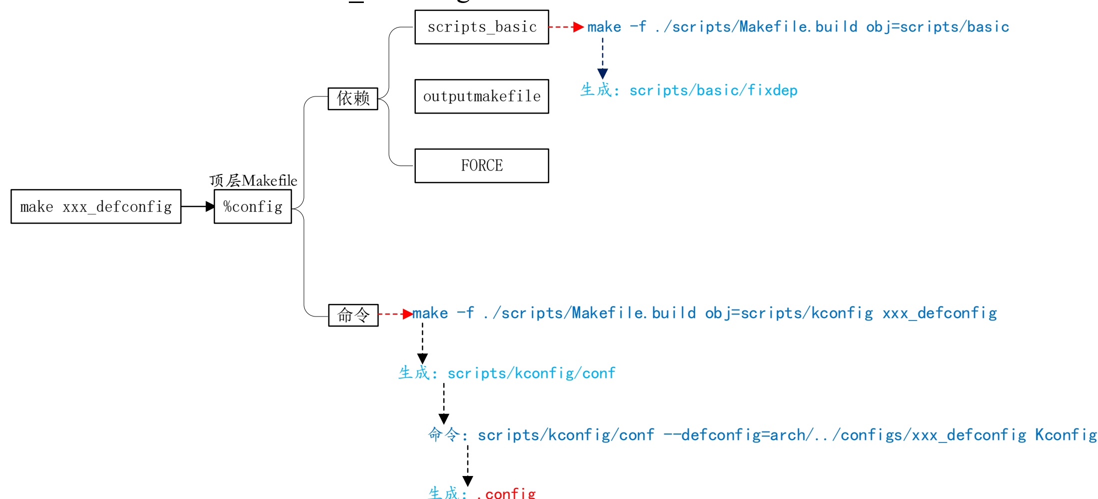


图 31.3.14.4 make xxx_defconfig 执行流程图

至此，make xxx_defconfig 就分析完了，接下来就要分析一下 u-boot.bin 是怎么生成的了。

31.3.15 make 过程

配置好 uboot 以后就可以直接 make 编译了，因为没有指明目标，所以会使用默认目标，主 Makefile 中的默认目标如下：

示例代码 31.3.15.1 顶层 Makefile 代码段

```
128 # That's our default target when none is given on the command line
129 PHONY := _all
130 _all:
```

目标_all 又依赖于 all，如下所示：

示例代码 31.3.15.2 顶层 Makefile 代码段

```
194 # If building an external module we do not care about the all: rule
195 # but instead _all depend on modules
```

```
196 PHONY += all
197 ifeq ($(KBUILD_EXTMOD),)
198 _all: all
199 else
200 _all: modules
201 endif
```

如果 KBUILD_EXTMOD 为空的话_all 依赖于 all。这里不编译模块，所以 KBUILD_EXTMOD 肯定为空，_all 的依赖就是 all。在主 Makefile 中 all 目标规则如下：

示例代码 31.3.15.2 顶层 Makefile 代码段

```
802 all:       $(ALL-y)
803 ifneq ($(CONFIG_SYS_GENERIC_BOARD),y)
804     @echo "===================== WARNING ======================"
805     @echo "Please convert this board to generic board."
806     @echo "Otherwise it will be removed by the end of 2014."
807     @echo "See doc/README.generic-board for further information"
808     @echo "===================================================="
809 endif
810 ifeq ($(CONFIG_DM_I2C_COMPAT),y)
811     @echo "===================== WARNING ======================"
812     @echo "This board uses CONFIG_DM_I2C_COMPAT. Please remove"
813     @echo "(possibly in a subsequent patch in your series)"
814     @echo "before sending patches to the mailing list."
815     @echo "===================================================="
816 endif
```

从 802 行可以看出，all 目标依赖$(ALL-y)，而在顶层 Makefile 中，ALL-y 如下：

示例代码 31.3.15.3 顶层 Makefile 代码段

```
730 # Always append ALL so that arch config.mk's can add custom ones
731 ALL-y += u-boot.srec u-boot.bin u-boot.sym System.map u-boot.cfg
binary_size_check
732
733 ALL-$(CONFIG_ONENAND_U_BOOT) += u-boot-onenand.bin
734 ifeq ($(CONFIG_SPL_FSL_PBL),y)
735 ALL-$(CONFIG_RAMBOOT_PBL) += u-boot-with-spl-pbl.bin
736 else
737 ifneq ($(CONFIG_SECURE_BOOT), y)
738 # For Secure Boot The Image needs to be signed and Header must also
739 # be included. So The image has to be built explicitly
740 ALL-$(CONFIG_RAMBOOT_PBL) += u-boot.pbl
741 endif
742 endif
743 ALL-$(CONFIG_SPL) += spl/u-boot-spl.bin
744 ALL-$(CONFIG_SPL_FRAMEWORK) += u-boot.img
745 ALL-$(CONFIG_TPL) += tpl/u-boot-tpl.bin
```

```
746 ALL-$(CONFIG_OF_SEPARATE) += u-boot.dtb
747 ifeq ($(CONFIG_SPL_FRAMEWORK),y)
748 ALL-$(CONFIG_OF_SEPARATE) += u-boot-dtb.img
749 endif
750 ALL-$(CONFIG_OF_HOSTFILE) += u-boot.dtb
751 ifneq ($(CONFIG_SPL_TARGET),)
752 ALL-$(CONFIG_SPL) += $(CONFIG_SPL_TARGET:"%"=%)
753 endif
754 ALL-$(CONFIG_REMAKE_ELF) += u-boot.elf
755 ALL-$(CONFIG_EFI_APP) += u-boot-app.efi
756 ALL-$(CONFIG_EFI_STUB) += u-boot-payload.efi
757 
758 ifneq ($(BUILD_ROM),)
759 ALL-$(CONFIG_X86_RESET_VECTOR) += u-boot.rom
760 endif
761 
762 # enable combined SPL/u-boot/dtb rules for tegra
763 ifeq ($(CONFIG_TEGRA)$(CONFIG_SPL),yy)
764 ALL-y += u-boot-tegra.bin u-boot-nodtb-tegra.bin
765 ALL-$(CONFIG_OF_SEPARATE) += u-boot-dtb-tegra.bin
766 endif
767 
768 # Add optional build target if defined in board/cpu/soc headers
769 ifneq ($(CONFIG_BUILD_TARGET),)
770 ALL-y += $(CONFIG_BUILD_TARGET:"%"=%)
771 endif
```

从示例代码代码 31.3.15.3 可以看出，ALL-y 包含 u-boot.srec、u-boot.bin、u-boot.sym、System.map、u-boot.cfg 和 binary_size_check 这几个文件。根据 uboot 的配置情况也可能包含其他的文件，比如：

```
ALL-$(CONFIG_ONENAND_U_BOOT) += u-boot-onenand.bin
```

CONFIG_ONENAND_U_BOOT 就是 uboot 中跟 ONENAND 配置有关的，如果我们使能了 ONENAND，那么在.config 配置文件中就会有“CONFIG_ONENAND_U_BOOT=y”这一句。相当于 CONFIG_ONENAND_U_BOOT 是个变量，这个变量的值为“y”，所以展开以后就是：

```
ALL-y += u-boot-onenand.bin
```

这个就是.config 里面的配置参数的含义，这些参数其实都是变量，后面跟着变量值，会在顶层 Makefile 或者其他 Makefile 中调用这些变量。

ALL-y 里面有个 u-boot.bin，这个就是我们最终需要的 uboot 二进制可执行文件，所作的所有工作就是为了它。在顶层 Makefile 中找到 u-boot.bin 目标对应的规则，如下所示：

示例代码 31.3.15.4 顶层 Makefile 代码段

```
825 ifeq ($(CONFIG_OF_SEPARATE),y)
826 u-boot-dtb.bin: u-boot-nodtb.bin dts/dt.dtb FORCE
827     $(call if_changed,cat)
828 
```

```
829 u-boot.bin: u-boot-dtb.bin FORCE
830     $(call if_changed,copy)
831 else
832 u-boot.bin: u-boot-nodtb.bin FORCE
833     $(call if_changed,copy)
834 endif
```

第 825 行判断 CONFIG_OF_SEPARATE 是否等于 y，如果相等，那条件就成立，在.config 中搜索“CONFIG_OF_SEPARAT”，没有找到，说明条件不成立。

第 832 行就是目标 u-boot.bin 的规则，目标 u-boot.bin 依赖于 u-boot-nodtb.bin，命令为$(call if_changed,copy)，这里调用了 if_changed，if_changed 是一个函数，这个函数在 scripts/Kbuild.include 中有定义，而顶层 Makefile 中会包含 scripts/Kbuild.include 文件，这个前面已经说过了。

if_changed 在 Kbuild.include 中的定义如下：

示例代码 31.3.15.5 Kbuild.include 代码段

```
226 ###
227 # if_changed     - execute command if any prerequisite is newer than
228 #                  target, or command line has changed
229 # if_changed_dep - as if_changed, but uses fixdep to reveal
230 #   dependencies including used config symbols
231 # if_changed_rule - as if_changed but execute rule instead
232 # See Documentation/kbuild/makefiles.txt for more info
233
234 ifneq ($(KBUILD_NOCMDDEP),1)
235 # Check if both arguments has same arguments. Result is empty
string if equal.
236 # User may override this check using make KBUILD_NOCMDDEP=1
237 arg-check = $(strip $(filter-out $(cmd_$(1)), $(cmd_$@)) \
238                     $(filter-out $(cmd_$@),   $(cmd_$(1))) )
239 else
240 arg-check = $(if $(strip $(cmd_$@)),,1)
241 endif
242
243 # Replace >$< with >$$< to preserve $ when reloading the .cmd file
244 # (needed for make)
245 # Replace >#< with >\#< to avoid starting a comment in the .cmd
file
246 # (needed for make)
247 # Replace >'< with >'\''< to be able to enclose the whole string in
'...'
248 # (needed for the shell)
249 make-cmd = $(call escsq,$(subst \#,\\\#,$(subst
$$,$$$$,$(cmd_$(1)))))
250
```

```
251 # Find any prerequisites that is newer than target or that does not
exist.
252 # PHONY targets skipped in both cases.
253 any-prereq = $(filter-out $(PHONY),$?) $(filter-out $(PHONY)
$(wildcard $^),$^)
254
255 # Execute command if command has changed or prerequisite(s) are
updated.
256 #
257 if_changed = $(if $(strip $(any-prereq) $(arg-check)), \
258     @set -e;                                            \
259     $(echo-cmd) $(cmd_$(1));                            \
260     printf '%s\n' 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd)
261
```

第 227 行为 if_changed 的描述，根据描述，在一些先决条件比目标新的时候，或者命令行有改变的时候，if_changed 就会执行一些命令。

第 257 行就是函数 if_changed，if_changed 函数引用的变量比较多，也比较绕，我们只需要知道它可以从 u-boot-nodtb.bin 生成 u-boot.bin 就行了。

既然 u-boot.bin 依赖于 u-boot-nodtb.bin，那么肯定要先生成 u-boot-nodtb.bin 文件，顶层 Makefile 中相关代码如下：

示例代码 31.3.15.6 顶层 Makefile 代码段

```
866 u-boot-nodtb.bin: u-boot FORCE
867     $(call if_changed,objcopy)
868     $(call DO_STATIC_RELA,$<,$@,$(CONFIG_SYS_TEXT_BASE))
869     $(BOARD_SIZE_CHECK)
```

目标 u-boot-nodtb.bin 又依赖于 u-boot，顶层 Makefile 中 u-boot 相关规则如下：

示例代码 31.3.15.7 顶层 Makefile 代码段

```
1170 u-boot:    $(u-boot-init) $(u-boot-main) u-boot.lds FORCE
1171     $(call if_changed,u-boot__)
1172 ifeq ($(CONFIG_KALLSYMS),y)
1173     $(call cmd,smap)
1174     $(call cmd,u-boot__) common/system_map.o
1175 endif
```

目标 u-boot 依赖于 u-boot_init、u-boot-main 和 u-boot.lds，u-boot_init 和 u-boot-main 是两个变量，在顶层 Makefile 中有定义，值如下：

示例代码 31.3.15.8 顶层 Makefile 代码段

```
678 u-boot-init := $(head-y)
679 u-boot-main := $(libs-y)
```

$(head-y)跟 CPU 架构有关，我们使用的是 ARM 芯片，所以 head-y 在 arch/arm/Makefile 中被指定为：

```
head-y := arch/arm/cpu/$(CPU)/start.o
```

根据 31.3.12 小节的分析，我们知道 CPU=armv7，因此 head-y 展开以后就是：

```
head-y := arch/arm/cpu/armv7/start.o
```

因此：

u-boot-init= arch/arm/cpu/armv7/start.o

$(libs-y)在顶层 Makefile 中被定义为 uboot 所有子目录下 build-in.o 的集合，代码如下：

示例代码 31.3.15.9 顶层 Makefile 代码段

```
620 libs-y += lib/
621 libs-$(HAVE_VENDOR_COMMON_LIB) += board/$(VENDOR)/common/
622 libs-$(CONFIG_OF_EMBED) += dts/
623 libs-y += fs/
624 libs-y += net/
625 libs-y += disk/
626 libs-y += drivers/
627 libs-y += drivers/dma/
628 libs-y += drivers/gpio/
629 libs-y += drivers/i2c/
......
660 libs-y += cmd/
661 libs-y += common/
662 libs-$(CONFIG_API) += api/
663 libs-$(CONFIG_HAS_POST) += post/
664 libs-y += test/
665 libs-y += test/dm/
666 libs-$(CONFIG_UT_ENV) += test/env/
667
668 libs-y += $(if $(BOARDDIR),board/$(BOARDDIR)/)
669
670 libs-y := $(sort $(libs-y))
671
672 u-boot-dirs := $(patsubst %/,%,$(filter %/, $(libs-y))) tools
examples
673
674 u-boot-alldirs  := $(sort $(u-boot-dirs)
$(patsubst %/,%,$(filter %/, $(libs-))))
675
676 libs-y     := $(patsubst %/, %/built-in.o, $(libs-y))
```

从上面的代码可以看出，libs-y 都是 uboot 各子目录的集合，最后：

libs-y := $(patsubst %/, %/built-in.o, $(libs-y))

这里调用了函数 patsubst，将 libs-y 中的“/”替换为”/built-in.o”，比如“drivers/dma/”就变为了“drivers/dma/built-in.o”，相当于将 libs-y 改为所有子目录中 built-in.o 文件的集合。那么 u-boot-main 就等于所有子目录中 built-in.o 的集合。

这个规则就相当于将以 u-boot.lds 为链接脚本，将 arch/arm/cpu/armv7/start.o 和各个子目录下的 built-in.o 链接在一起生成 u-boot。

u-boot.lds 的规则如下：

示例代码 31.3.15.10 顶层 Makefile 代码段

```
u-boot.lds: $(LDSCRIPT) prepare FORCE
    $(call if_changed_dep,cpp_lds)
```

接下来的重点就是各子目录下的 built-in.o 是怎么生成的，以 drivers/gpio/built-in.o 为例，在 drivers/gpio/目录下会有个名为.built-in.o.cmd 的文件，此文件内容如下：

示例代码 31.3.15.11 drivers/gpio/.built-in.o.cmd 代码

```
1 cmd_drivers/gpio/built-in.o := arm-linux-gnueabihf-ld.bfd    -r -o
drivers/gpio/built-in.o drivers/gpio/mxc_gpio.o
```

从命令“cmd_drivers/gpio/built-in.o”可以看出，drivers/gpio/built-in.o 这个文件是使用 ld 命令由文件 drivers/gpio/mxc_gpio.o 生成而来的，mxc_gpio.o 是 mxc_gpio.c 编译生成的.o 文件，这个是 NXP 的 I.MX 系列的 GPIO 驱动文件。这里用到了 ld 的“-r”参数，参数含义如下：

-r –relocateable: 产生可重定向的输出，比如，产生一个输出文件它可再次作为‘ld’的输入，这经常被叫做“部分链接”，当我们需要将几个小的.o 文件链接成为一个.o 文件的时候，需要使用此选项。

最终将各个子目录中的 built-in.o 文件链接在一起就形成了 u-boot，使用如下命令编译 uboot 就可以看到链接的过程：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_ddr512_emmc_defconfig V=1
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- V=1
```

编译的时候会有如图 31.3.15.1 所示内容输出：

```
arm-linux-gnueabihf-ld.bfd   -pie  --gc-sections -Bstatic -Ttext 0x87800000 -o u-boot -T u-boot.lds arch/arm/cpu/armv7/start.o --start-group  arch/arm/cpu/built-in.o  arch/arm/cpu/armv7/built-in.o  arch/arm/imx-common/built-in.o  arch/arm/lib/built-in.o  board/freescale/common/built-in.o  board/freescale/mx6ull_alientek_emmc/built-in.o  cmd/built-in.o  common/built-in.o  disk/built-in.o  drivers/built-in.o  drivers/dma/built-in.o  drivers/gpio/built-in.o  drivers/i2c/built-in.o  drivers/mmc/built-in.o  drivers/mtd/built-in.o  drivers/mtd/onenand/built-in.o  drivers/mtd/spi/built-in.o  drivers/net/built-in.o  drivers/net/phy/built-in.o  drivers/pci/built-in.o  drivers/power/built-in.o  drivers/power/battery/built-in.o  drivers/power/fuel_gauge/built-in.o  drivers/power/mfd/built-in.o  drivers/power/pmic/built-in.o  drivers/power/regulator/built-in.o  drivers/serial/built-in.o  drivers/spi/built-in.o  drivers/usb/dwc3/built-in.o  drivers/usb/emul/built-in.o  drivers/usb/eth/built-in.o  drivers/usb/gadget/built-in.o  drivers/usb/gadget/udc/built-in.o  drivers/usb/host/built-in.o  drivers/usb/musb-new/built-in.o  drivers/usb/musb/built-in.o  drivers/usb/phy/built-in.o  drivers/usb/ulpi/built-in.o  fs/built-in.o  lib/built-in.o  net/built-in.o  test/built-in.o  test/dm/built-in.o --end-group arch/arm/lib/eabi_compat.o  -L /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/4.9.4 -lgcc -Map u-boot.map
```

图 31.3.15.1 编译内容输出

将其整理一下，内容如下：

```
arm-linux-gnueabihf-ld.bfd    -pie   --gc-sections -Bstatic -Ttext 0x87800000 \
-o u-boot -T u-boot.lds \
arch/arm/cpu/armv7/start.o \
--start-group   arch/arm/cpu/built-in.o \
arch/arm/cpu/armv7/built-in.o \
arch/arm/imx-common/built-in.o \
arch/arm/lib/built-in.o \
board/freescale/common/built-in.o \
board/freescale/mx6ull_alientek_emmc/built-in.o \
cmd/built-in.o    \
common/built-in.o    \
disk/built-in.o    \
drivers/built-in.o    \
drivers/dma/built-in.o    \
drivers/gpio/built-in.o    \
```

```
……
drivers/spi/built-in.o    \
drivers/usb/dwc3/built-in.o    \
drivers/usb/emul/built-in.o    \
drivers/usb/eth/built-in.o    \
drivers/usb/gadget/built-in.o    \
drivers/usb/gadget/udc/built-in.o    \
drivers/usb/host/built-in.o    \
drivers/usb/musb-new/built-in.o    \
drivers/usb/musb/built-in.o    \
drivers/usb/phy/built-in.o    \
drivers/usb/ulpi/built-in.o    \
fs/built-in.o    \
lib/built-in.o    \
net/built-in.o    \
test/built-in.o    \
test/dm/built-in.o \
--end-group arch/arm/lib/eabi_compat.o    \
-L /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/4.9.4 -lgcc -Map u-boot.map
```

可以看出最终是用 arm-linux-gnueabihf-ld.bfd 命令将 arch/arm/cpu/armv7/start.o 和其他众多的 built_in.o 链接在一起，形成 u-boot。

目标 all 除了 u-boot.bin 以外还有其他的依赖，比如 u-boot.srec 、u-boot.sym 、System.map、u-boot.cfg 和 binary_size_check 等等，这些依赖的生成方法和 u-boot.bin 很类似，大家自行查看一下顶层 Makefile，我们就不详细的讲解了。

总结一下“make”命令的流程，如图 31.3.15.2 所示：

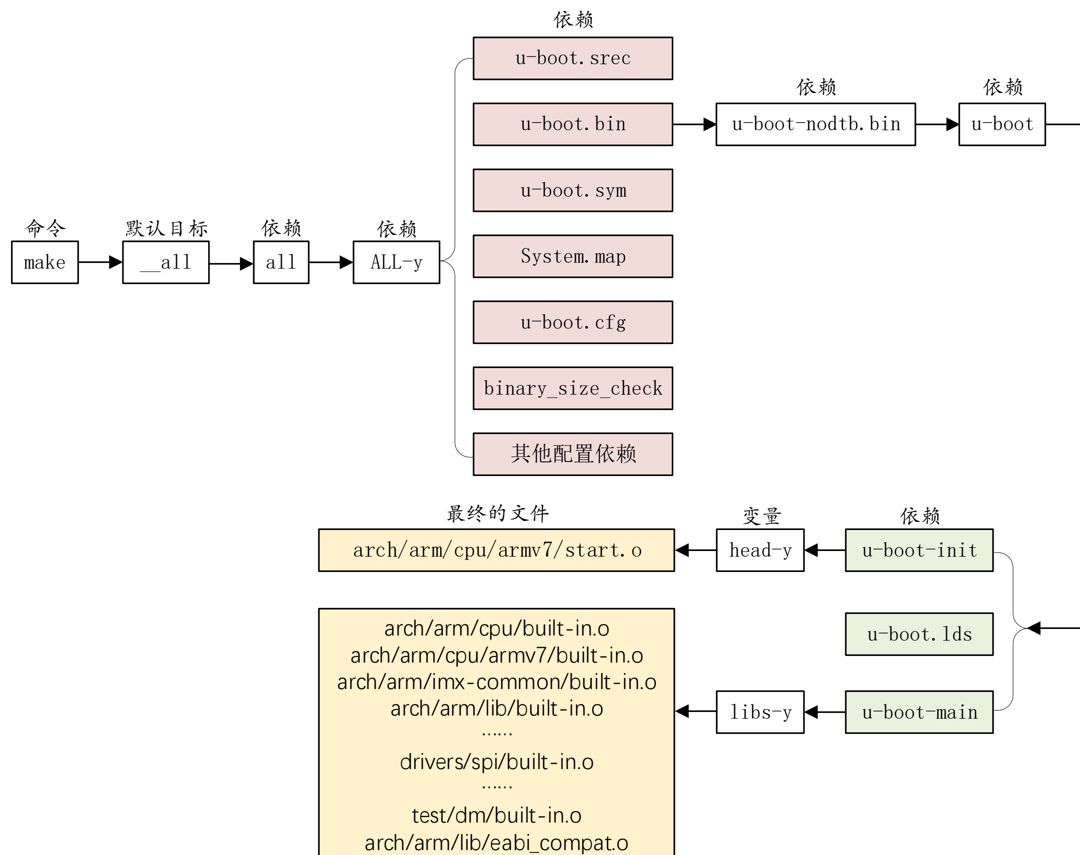


图 31.3.15.2 make 命令流程

图 31.3.15.2 就是“make”命令的执行流程，关于 uboot 的顶层 Makefile 就分析到这里，重点是“make xxx_defconfig”和“make”这两个命令的执行流程：

make xxx_defconfig：用于配置 uboot，这个命令最主要的目的就是生成.config 文件。

make：用于编译 uboot，这个命令的主要工作就是生成二进制的 u-boot.bin 文件和其他的一些与 uboot 有关的文件，比如 u-boot.imx 等等。

关于 uboot 的顶层 Makefile 就分析到这里，有些内容我们没有详细、深入的去研究，因为我们的重点是使用 uboot，而不是 uboot 的研究者，我们要做的是缕清 uboot 的流程。至于更具体的实现，有兴趣的可以参考一下其他资料。

第三十二章 U-Boot 启动流程详解

上一章我们详细的分析了 uboot 的顶层 Makefile，理清了 uboot 的编译流程。本章我们来详细的分析一下 uboot 的启动流程，理清 uboot 是如何启动的。通过对 uboot 启动流程的梳理，我们就可以掌握一些外设是在哪里被初始化的，这样当我们需要修改这些外设驱动的时候就会心里有数。另外，通过分析 uboot 的启动流程可以了解 Linux 内核是如何被启动的。

32.1 链接脚本 u-boot.lds 详解

要分析 uboot 的启动流程，首先要找到“入口”，找到第一行程序在哪里。程序的链接是由链接脚本来决定的，所以通过链接脚本可以找到程序的入口。如果没有编译过 uboot 的话链接脚本为 arch/arm/cpu/u-boot.lds。但是这个不是最终使用的链接脚本，最终的链接脚本是在这个链接脚本的基础上生成的。编译一下 uboot，编译完成以后就会在 uboot 根目录下生成 u-boot.lds 文件，如图 32.1.1 所示：

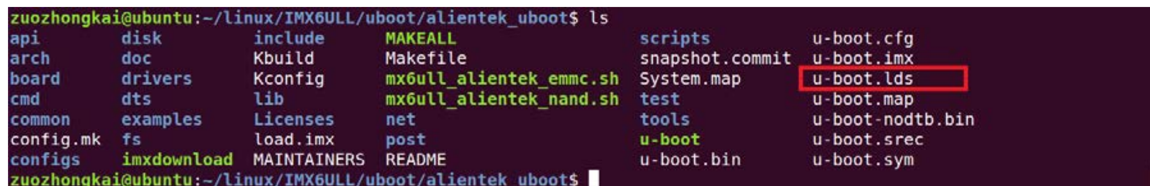


图 32.1.1 链接脚本

只有编译 u-boot 以后才会在根目录下出现 u-boot.lds 文件！
只有编译 u-boot 以后才会在根目录下出现 u-boot.lds 文件！
只有编译 u-boot 以后才会在根目录下出现 u-boot.lds 文件！

打开 u-boot.lds，内容如下：

示例代码 32.1.1 u-boot.lds 文件代码

```
1  OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
2  OUTPUT_ARCH(arm)
3  ENTRY(_start)
4  SECTIONS
5  {
6   . = 0x00000000;
7   . = ALIGN(4);
8   .text :
9   {
10   *(.__image_copy_start)
11   *(.vectors)
12   arch/arm/cpu/armv7/start.o (.text*)
13   *(.text*)
14  }
15  . = ALIGN(4);
16  .rodata : { *(SORT_BY_ALIGNMENT(SORT_BY_NAME(.rodata*))) }
17  . = ALIGN(4);
18  .data : {
19   *(.data*)
20  }
21  . = ALIGN(4);
22  . = .;
23  . = ALIGN(4);
24  .u_boot_list : {
25   KEEP(*(SORT(.u_boot_list*)));
```

```
26 }
27 . = ALIGN(4);
28 .image_copy_end :
29 {
30  *(.__image_copy_end)
31 }
32 .rel_dyn_start :
33 {
34  *(.__rel_dyn_start)
35 }
36 .rel.dyn : {
37  *(.rel*)
38 }
39 .rel_dyn_end :
40 {
41  *(.__rel_dyn_end)
42 }
43 .end :
44 {
45  *(.__end)
46 }
47 _image_binary_end = .;
48 . = ALIGN(4096);
49 .mmutable : {
50  *(.mmutable)
51 }
52 .bss_start __rel_dyn_start (OVERLAY) : {
53  KEEP(*(.__bss_start));
54  __bss_base = .;
55 }
56 .bss __bss_base (OVERLAY) : {
57  *(.bss*)
58   . = ALIGN(4);
59   __bss_limit = .;
60 }
61 .bss_end __bss_limit (OVERLAY) : {
62  KEEP(*(.__bss_end));
63 }
64 .dynsym _image_binary_end : { *(.dynsym) }
65 .dynbss : { *(.dynbss) }
66 .dynstr : { *(.dynstr*) }
67 .dynamic : { *(.dynamic*) }
68 .plt : { *(.plt*) }
```

```
69 .interp : { *(.interp*) }
70 .gnu.hash : { *(.gnu.hash) }
71 .gnu : { *(.gnu*) }
72 .ARM.exidx : { *(.ARM.exidx*) }
73 .gnu.linkonce.armexidx : { *(.gnu.linkonce.armexidx.*) }
74 }
```

第 3 行为代码当前入口点：_start， _start 在文件 arch/arm/lib/vectors.S 中有定义，如图 32.1.2 所示：

```
vectors.S
26 .globl _start
27
28 /*
29  *************************************************************************
30  *
31  * Vectors have their own section so linker script can map them easily
32  *
33  *************************************************************************
34  */
35
36     .section ".vectors", "ax"
37
38 /*
39  *************************************************************************
40  *
41  * Exception vectors as described in ARM reference manuals
42  *
43  * Uses indirect branch to allow reaching handlers anywhere in memory.
44  *
45  *************************************************************************
46  */
47
48 _start:
49
50 #ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
51     .word   CONFIG_SYS_DV_NOR_BOOT_CFG
52 #endif
53
54     b   reset
55     ldr pc, _undefined_instruction
56     ldr pc, _software_interrupt
57     ldr pc, _prefetch_abort
58     ldr pc, _data_abort
59     ldr pc, _not_used
60     ldr pc, _irq
61     ldr pc, _fiq
```

图 32.1.2 _start 入口

从图 32.1.1 可以看出，_start 后面就是中断向量表，从图中的“.section ".vectors", "ax”可以得到，此代码存放在.vectors 段里面。

第 10 行，使用如下命令在 uboot 中查找“__image_copy_start”：

```
grep -nR "__image_copy_start"
```

搜索结果如图 32.1.3 所示：

```
arch/arc/lib/relocate.c:17:      memcpy((void *)gd->relocaddr, (void *)&__image_copy_start, len);
arch/arc/lib/relocate.c:48:              if (offset_ptr_rom >= (Elf32_Addr *)&__image_copy_start &&
arch/arc/lib/relocate.c:71:                      if (val >= (unsigned int)&__image_copy_start && val <=
arch/arc/cpu/u-boot.lds:15:              *(.__image_copy_start)
u-boot.lds:10:  *(.__image_copy_start)
u-boot.map:931: *(.__image_copy_start)
u-boot.map:932:  .__image_copy_start
u-boot.map:934:                 0x0000000087800000                __image_copy_start
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/uboot-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 32.1.3 查找结果

打开 u-boot.map，找到如图 32.1.4 所示位置：

```
u-boot.map
926  段 .text 的地址设置为 0x87800000
927                 0x0000000000000000                . = 0x0
928                 0x0000000000000000                . = ALIGN (0x4)
929
930  .text          0x0000000087800000    0x3e94c
931   *(.__image_copy_start)
932   .__image_copy_start
933                 0x0000000087800000       0x0 arch/arm/lib/built-in.o
934                 0x0000000087800000                __image_copy_start
935   *(.vectors)
936   .vectors      0x0000000087800000     0x300 arch/arm/lib/built-in.o
937                 0x0000000087800000                _start
938                 0x0000000087800020                _undefined_instruction
939                 0x0000000087800024                _software_interrupt
940                 0x0000000087800028                _prefetch_abort
941                 0x000000008780002c                _data_abort
942                 0x0000000087800030                _not_used
943                 0x0000000087800034                _irq
944                 0x0000000087800038                _fiq
945                 0x0000000087800040                IRQ_STACK_START_IN
```

图 32.1.4 u-boot.map

u-boot.map 是 uboot 的映射文件，可以从此文件看到某个文件或者函数链接到了哪个地址，从图 32.1.4 的 932 行可以看到__image_copy_start 为 0X87800000，而.text 的起始地址也是 0X87800000。

第 11 行是 vectors 段，vectors 段保存中断向量表，从图 32.1.2 中我们知道了 vectors.S 的代码是存在 vectors 段中的。从图 32.1.4 可以看出，vectors 段的起始地址也是 0X87800000，说明整个 uboot 的起始地址就是 0X87800000，这也是为什么我们裸机例程的链接起始地址选择 0X87800000 了，目的就是为了和 uboot 一致。

第 12 行将 arch/arm/cpu/armv7/start.s 编译出来的代码放到中断向量表后面。

第 13 行为 text 段，其他的代码段就放到这里

在 u-boot.lds 中有一些跟地址有关的“变量”需要我们注意一下，后面分析 u-boot 源码的时候会用到，这些变量要最终编译完成才能确定的!!! 比如我编译完成以后这些“变量”的值如表 32.1.1 所示：

变量	数值	描述
__image_copy_start	0x87800000	uboot 拷贝的首地址
__image_copy_end	0x8785dd54	uboot 拷贝的结束地址
__rel_dyn_start	0x8785dd54	
__rel_dyn_end	0x878668f4	

_image_binary_end	0x878668f4	
__bss_start	0x8785dd54	
__bss_end	0x878a8e74	

表 32.1.1 uboot 相关变量表

表 32.1.1 中的“变量”值可以在 u-boot.map 文件中查找，表 32.1.1 中除了__image_copy_start 以外，其他的变量值每次编译的时候可能会变化，如果修改了 uboot 代码、修改了 uboot 配置、选用不同的优化等级等等都会影响到这些值。所以，一切以实际值为准！

32.2 U-Boot 启动流程详解

32.2.1 reset 函数源码详解

从 u-boot.lds 中我们已经知道了入口点是 arch/arm/lib/vectors.S 文件中的_start，代码如下：

示例代码 32.2.1.1 vectors.S 代码段

```
38 /*
39  *************************************************************
40  *
41  * Exception vectors as described in ARM reference manuals
42  *
43  * Uses indirect branch to allow reaching handlers anywhere in
44  * memory.
45  **************************************************************
46  */
47
48 _start:
49
50 #ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
51  .word   CONFIG_SYS_DV_NOR_BOOT_CFG
52 #endif
53
54      b   reset
55      ldr pc, _undefined_instruction
56      ldr pc, _software_interrupt
57      ldr pc, _prefetch_abort
58      ldr pc, _data_abort
59      ldr pc, _not_used
60      ldr pc, _irq
61      ldr pc, _fiq
```

第 48 行_start 开始的是中断向量表，其中 54~61 行就是中断向量表，和我们裸机例程里面一样。

54 行跳转到 reset 函数里面，reset 函数在 arch/arm/cpu/armv7/start.S 里面，代码如下：

示例代码 32.2.1.2 start.S 代码段

```
22 /*****************************************************************
23  *
24  * Startup Code (reset vector)
```

```
25  *
26  * Do important init only if we don't start from memory!
27  * Setup memory and board specific bits prior to relocation.
28  * Relocate armboot to ram. Setup stack.
29  *
30  *********************************************************************/
31 
32  .globl  reset
33  .globl  save_boot_params_ret
34 
35 reset:
36     /* Allow the board to save important registers */
37     b   save_boot_params
```

第 35 行就是 reset 函数。

第 37 行从 reset 函数跳转到了 save_boot_params 函数，而 save_boot_params 函数同样定义在 start.S 里面，定义如下：

示例代码 32.2.1.3 start.S 代码段

```
91  /*********************************************************************
92   *
93   * void save_boot_params(u32 r0, u32 r1, u32 r2, u32 r3)
94   *  __attribute__((weak));
95   *
96   * Stack pointer is not yet initialized at this moment
97   * Don't save anything to stack even if compiled with -O0
98   *
99   *********************************************************************/
100 ENTRY(save_boot_params)
101     b   save_boot_params_ret        @ back to my caller
```

save_boot_params 函数也是只有一句跳转语句，跳转到 save_boot_params_ret 函数，save_boot_params_ret 函数代码如下：

示例代码 32.2.1.4 start.S 代码段

```
38 save_boot_params_ret:
39     /*
40     * disable interrupts (FIQ and IRQ), also set the cpu to SVC32
41     * mode, except if in HYP mode already
42     */
43     mrs     r0, cpsr
44     and     r1, r0,     #0x1f   @ mask mode bits
45     teq     r1,         #0x1a   @ test for HYP mode
46     bicne   r0, r0,     #0x1f   @ clear all mode bits
47     orrne   r0, r0,     #0x13   @ set SVC mode
48     orr     r0, r0,     #0xc0   @ disable FIQ and IRQ
49     msr     cpsr,r0
```

第 43 行，读取寄存器 cpsr 中的值，并保存到 r0 寄存器中。

第 44 行，将寄存器 r0 中的值与 0X1F 进行与运算，结果保存到 r1 寄存器中，目的就是提取 cpsr 的 bit0~bit4 这 5 位，这 5 位为 M4 M3 M2 M1 M0，M[4:0]这五位用来设置处理器的工作模式，如表 32.2.1.1 所示：

M[4:0]	模式
10000	User(usr)
10001	FIQ(fiq)
10010	IRQ(irq)
10011	Supervisor(svc)
10110	Monitor(mon)
10111	Abort(abt)
11010	Hyp(hyp)
11011	Undefined(und)
11111	System(sys)

表 32.2.1.1 Cortex-A7 工作模式

第 45 行，判断 r1 寄存器的值是否等于 0X1A(0b11010)，也就是判断当前处理器模式是否处于 Hyp 模式。

第 46 行，如果 r1 和 0X1A 不相等，也就是 CPU 不处于 Hyp 模式的话就将 r0 寄存器的 bit0~5 进行清零，其实就是清除模式位

第 47 行，如果处理器不处于 Hyp 模式的话就将 r0 的寄存器的值与 0x13 进行或运算，0x13=0b10011，也就是设置处理器进入 SVC 模式。

第 48 行，r0 寄存器的值再与 0xC0 进行或运算，那么 r0 寄存器此时的值就是 0xD3，cpsr 的 I 为和 F 位分别控制 IRQ 和 FIQ 这两个中断的开关，设置为 1 就关闭了 FIQ 和 IRQ！

第 49 行，将 r0 寄存器写回到 cpsr 寄存器中。完成设置 CPU 处于 SVC32 模式，并且关闭 FIQ 和 IRQ 这两个中断。

继续执行执行下面的代码：

示例代码 32.2.1.5 start.S 代码段

```
51 /*
52  * Setup vector:
53  * (OMAP4 spl TEXT_BASE is not 32 byte aligned.
54  * Continue to use ROM code vector only in OMAP4 spl)
55  */
56 #if !(defined(CONFIG_OMAP44XX) && defined(CONFIG_SPL_BUILD))
57  /* Set V=0 in CP15 SCTLR register - for VBAR to point to vector */
58     mrc p15, 0, r0, c1, c0, 0   @ Read CP15 SCTLR Register
59     bic r0, #CR_V               @ V = 0
60     mcr p15, 0, r0, c1, c0, 0   @ Write CP15 SCTLR Register
61 
62     /* Set vector address in CP15 VBAR register */
63     ldr r0, =_start
64     mcr p15, 0, r0, c12, c0, 0  @Set VBAR
65 #endif
```

第 56 行，如果没有定义 CONFIG_OMAP44XX 和 CONFIG_SPL_BUILD 的话条件件成立，

此处条件成立。

第 58 行读取 CP15 中 c1 寄存器的值到 r0 寄存器中，根据 17.1.4 小节可知，这里是读取 SCTLR 寄存器的值。

第 59 行，CR_V 在 arch/arm/include/asm/system.h 中有如下所示定义：

```
#define CR_V (1 << 13) /* Vectors relocated to 0xffff0000  */
```

因此这一行的目的就是清除 SCTLR 寄存器中的 bit13，SCTLR 寄存器结构如图 32.2.1.1 所示：

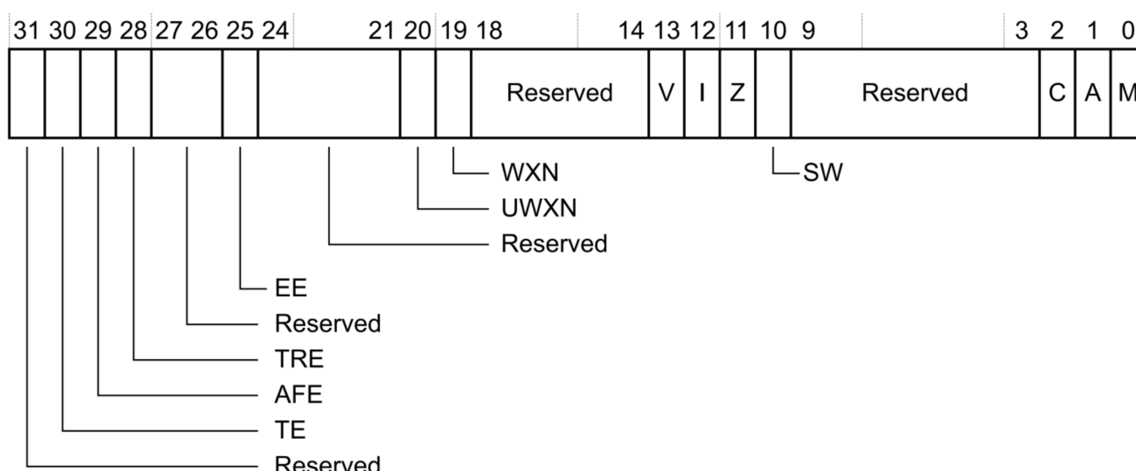


图 32.2.1.1 SCTLR 寄存器结构图

从图 32.2.1.1 可以看出，bit13 为 V 位，此位是向量表控制位，当为 0 的时候向量表基地址为 0X00000000，软件可以重定位向量表。为 1 的时候向量表基地址为 0XFFFF0000，软件不能重定位向量表。这里将 V 清零，目的就是为了接下来的向量表重定位，这个我们在第十七章有过详细的介绍了。

第 60 行将 r0 寄存器的值重写写入到寄存器 SCTLR 中。

第 63 行设置 r0 寄存器的值为_start，_start 就是整个 uboot 的入口地址，其值为 0X87800000，相当于 uboot 的起始地址，因此 0x87800000 也是向量表的起始地址。

第 64 行将 r0 寄存器的值(向量表值)写入到 CP15 的 c12 寄存器中，也就是 VBAR 寄存器。因此第 58~64 行就是设置向量表重定位的。

代码继续往下执行：

示例代码 32.2.1.6 start.S 代码段

```
67  /* the mask ROM code should have PLL and others stable */
68 #ifndef CONFIG_SKIP_LOWLEVEL_INIT
69      bl  cpu_init_cp15
70      bl  cpu_init_crit
71 #endif
72
73  bl  _main
```

第 68 行如果没有定义 CONFIG_SKIP_LOWLEVEL_INIT 的话条件成立。我们没有定义 CONFIG_SKIP_LOWLEVEL_INIT，因此条件成立，执行下面的语句。

示例代码 32.2.1.6 中的内容比较简单，就是分别调用函数 cpu_init_cp15、cpu_init_crit 和 _main。

函数 cpu_init_cp15 用来设置 CP15 相关的内容，比如关闭 MMU 啥的，此函数同样在 start.S

文件中定义的，代码如下：

示例代码 32.2.1.7 start.S 代码段

```
105 /*************************************************************************
106  *
107  * cpu_init_cp15
108  *
109  * Setup CP15 registers (cache, MMU, TLBs). The I-cache is turned on
110  * unless CONFIG_SYS_ICACHE_OFF is defined.
111  *
112  *************************************************************************/
113 ENTRY(cpu_init_cp15)
114     /*
115      * Invalidate L1 I/D
116      */
117     mov r0, #0                  @ set up for MCR
118     mcr p15, 0, r0, c8, c7, 0   @ invalidate TLBs
119     mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
120     mcr p15, 0, r0, c7, c5, 6   @ invalidate BP array
121     mcr p15, 0, r0, c7, c10,4   @ DSB
122     mcr p15, 0, r0, c7, c5, 4   @ ISB
123 
124     /*
125      * disable MMU stuff and caches
126      */
127     mrc p15, 0, r0, c1, c0, 0
128     bic r0, r0, #0x00002000 @ clear bits 13 (--V-)
129     bic r0, r0, #0x00000007 @ clear bits 2:0 (-CAM)
130     orr r0, r0, #0x00000002 @ set bit 1 (--A-) Align
131     orr r0, r0, #0x00000800 @ set bit 11 (Z---) BTB
132 #ifdef CONFIG_SYS_ICACHE_OFF
133     bic r0, r0, #0x00001000 @ clear bit 12 (I) I-cache
134 #else
135     orr r0, r0, #0x00001000 @ set bit 12 (I) I-cache
136 #endif
137     mcr p15, 0, r0, c1, c0, 0
138 
......
255 
256     mov pc, r5          @ back to my caller
257 ENDPROC(cpu_init_cp15)
```

函数 cpu_init_cp15 都是一些和 CP15 有关的内容，我们不用关心，有兴趣的可以详细的看一下。

函数 cpu_init_crit 也在是定义在 start.S 文件中，函数内容如下：

示例代码 32.2.1.8 start.S 代码段

```
260 /*************************************************************
261  *
262  * CPU_init_critical registers
263  *
264  * setup important registers
265  * setup memory timing
266  *
267  *************************************************************/
268 ENTRY(cpu_init_crit)
269     /*
270      * Jump to board specific initialization...
271      * The Mask ROM will have already initialized
272      * basic memory. Go here to bump up clock rate and handle
273      * wake up conditions.
274      */
275     b   lowlevel_init      @ go setup pll,mux,memory
276 ENDPROC(cpu_init_crit)
```

可以看出函数 cpu_init_crit 内部仅仅是调用了函数 lowlevel_init，接下来就是详细的分析一下 lowlevel_init 和_main 这两个函数。

32.2.2 lowlevel_init 函数详解

函数 lowlevel_init 在文件 arch/arm/cpu/armv7/lowlevel_init.S 中定义，内容如下：

示例代码 32.2.2.1 lowlevel_init.S 代码段

```
14 #include <asm-offsets.h>
15 #include <config.h>
16 #include <linux/linkage.h>
17
18 ENTRY(lowlevel_init)
19     /*
20     * Setup a temporary stack. Global data is not available yet.
21     */
22     ldr sp, =CONFIG_SYS_INIT_SP_ADDR
23     bic sp, sp, #7       /* 8-byte alignment for ABI compliance */
24 #ifdef CONFIG_SPL_DM
25     mov r9, #0
26 #else
27     /*
28     * Set up global data for boards that still need it. This will be
29     * removed soon.
30     */
31 #ifdef CONFIG_SPL_BUILD
32     ldr r9, =gdata
```

```
33 #else
34      sub sp, sp, #GD_SIZE
35      bic sp, sp, #7
36      mov r9, sp
37 #endif
38 #endif
39      /*
40      * Save the old lr(passed in ip) and the current lr to stack
41      */
42      push    {ip, lr}
43
44      /*
45      * Call the very early init function. This should do only the
46      * absolute bare minimum to get started. It should not:
47      *
48      * - set up DRAM
49      * - use global_data
50      * - clear BSS
51      * - try to start a console
52      *
53      * For boards with SPL this should be empty since SPL can do all
54      * of this init in the SPL board_init_f() function which is
55      * called immediately after this.
56      */
57      bl  s_init
58      pop {ip, pc}
59 ENDPROC(lowlevel_init)
```

第 22 行设置 sp 指向 CONFIG_SYS_INIT_SP_ADDR，CONFIG_SYS_INIT_SP_ADDR 在 include/configs/mx6ullevk.h 文件中，在 mx6ullevk.h 中有如下所示定义：

示例代码 32.2.2.2 mx6ullevk.h 代码段

```
234 #define CONFIG_SYS_INIT_RAM_ADDR    IRAM_BASE_ADDR
235 #define CONFIG_SYS_INIT_RAM_SIZE    IRAM_SIZE
236
237 #define CONFIG_SYS_INIT_SP_OFFSET \
238     (CONFIG_SYS_INIT_RAM_SIZE - GENERATED_GBL_DATA_SIZE)
239 #define CONFIG_SYS_INIT_SP_ADDR \
240     (CONFIG_SYS_INIT_RAM_ADDR + CONFIG_SYS_INIT_SP_OFFSET)
```

示例代码 32.2.2.2 中的 IRAM_BASE_ADDR 和 IRAM_SIZE 在文件 arch/arm/include/asm/arch-mx6/imx-regs.h 中有定义，如下所示，其实就是 IMX6UL/IM6ULL 内部 ocram 的首地址和大小。

示例代码 32.2.2.3 imx-regs.h 代码段

```
71  #define IRAM_BASE_ADDR              0x00900000
......
```

```
408 #if !(defined(CONFIG_MX6SX) || defined(CONFIG_MX6UL) || \
409     defined(CONFIG_MX6SLL) || defined(CONFIG_MX6SL))
410 #define IRAM_SIZE                   0x00040000
411 #else
412 #define IRAM_SIZE                   0x00020000
413 #endif
```

如果 408 行的条件成立的话 IRAM_SIZE=0X40000，当定义了 CONFIG_MX6SX、CONFIG_MX6U、CONFIG_MX6SLL 和 CONFIG_MX6SL 中的任意一个的话条件就不成立，在.config 中定义了 CONFIG_MX6UL，所以条件不成立，因此 IRAM_SIZE=0X20000=128KB。

结合示例代码 32.2.2.2，可以得到如下值：

CONFIG_SYS_INIT_RAM_ADDR = IRAM_BASE_ADDR = 0x00900000。

CONFIG_SYS_INIT_RAM_SIZE = 0x00020000 =128KB。

还需要知道 GENERATED_GBL_DATA_SIZE 的值，在文件 include/generated/generic-asm-offsets.h 中有定义，如下：

示例代码 32.2.2.4 generic-asm-offsets.h 代码段

```
1  #ifndef __GENERIC_ASM_OFFSETS_H__
2  #define __GENERIC_ASM_OFFSETS_H__
3  /*
4   * DO NOT MODIFY.
5   *
6   * This file was generated by Kbuild
7   */
8
9  #define GENERATED_GBL_DATA_SIZE  256
10 #define GENERATED_BD_INFO_SIZE   80
11 #define GD_SIZE                  248
12 #define GD_BD                    0
13 #define GD_MALLOC_BASE           192
14 #define GD_RELOCADDR             48
15 #define GD_RELOC_OFF             68
16 #define GD_START_ADDR_SP         64
17
18 #endif
```

GENERATED_GBL_DATA_SIZE=256，GENERATED_GBL_DATA_SIZE 的含义为 (sizeof(struct global_data) + 15) & ~15 。

综上所述，CONFIG_SYS_INIT_SP_ADDR 值如下：

CONFIG_SYS_INIT_SP_OFFSET = 0x00020000 – 256 = 0x1FF00。

CONFIG_SYS_INIT_SP_ADDR = 0x00900000 + 0X1FF00 = 0X0091FF00,

结果如下图所示：

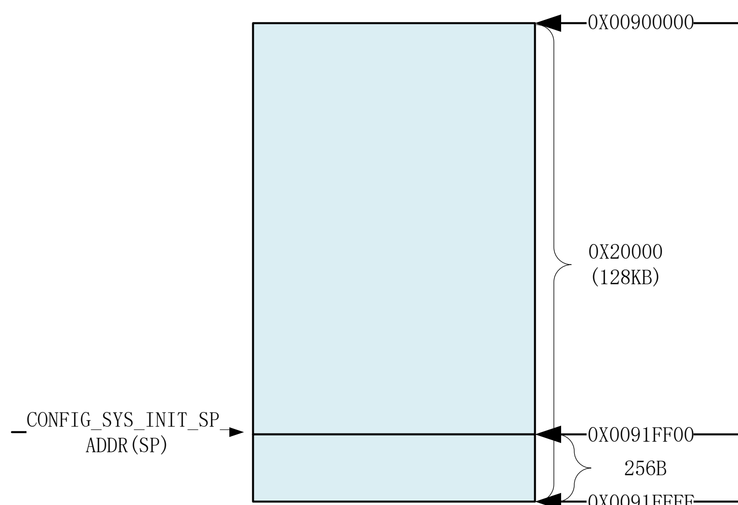


图 32.2.2.1 sp 值

此时 sp 指向 0X91FF00，这属于 IMX6UL/IMX6ULL 的内部 ram。

继续回到文件 lowlevel_init.S，第 23 行对 sp 指针做 8 字节对齐处理！

第 34 行，sp 指针减去 GD_SIZE，GD_SIZE 同样在 generic-asm-offsets.h 中定了，大小为 248，见示例代码 32.2.2.4 第 11 行。

第 35 行对 sp 做 8 字节对齐，此时 sp 的地址为 0X0091FF00-248=0X0091FE08，此时 sp 位置如图 32.2.2.2 所示：

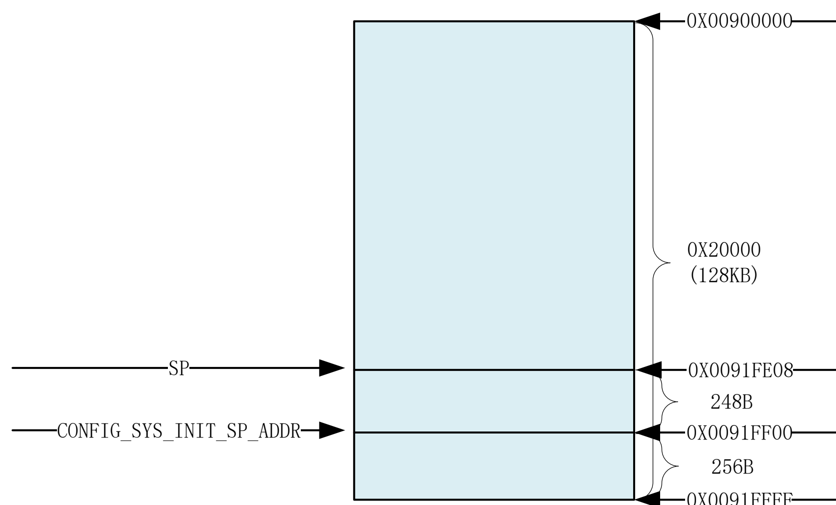


图 32.2.2.2 sp 值

第 36 行将 sp 地址保存在 r9 寄存器中。

第 42 行将 ip 和 lr 压栈

第 57 行调用函数 s_init，得，又来了一个函数。

第 58 行将第 36 行入栈的 ip 和 lr 进行出栈，并将 lr 赋给 pc。

32.2.3 s_init 函数详解

在上一小节中，我们知道 lowlevel_init 函数后面会调用 s_init 函数，s_init 函数定义在文件 arch/arm/cpu/armv7/mx6/soc.c 中，如下所示：

示例代码 32.2.3.1 soc.c 代码段

```
808 void s_init(void)
809 {
810     struct anatop_regs *anatop = (struct anatop_regs
*)ANATOP_BASE_ADDR;
811     struct mxc_ccm_reg *ccm = (struct mxc_ccm_reg *)CCM_BASE_ADDR;
812     u32 mask480;
813     u32 mask528;
814     u32 reg, periph1, periph2;
815 
816     if (is_cpu_type(MXC_CPU_MX6SX) || is_cpu_type(MXC_CPU_MX6UL) ||
817         is_cpu_type(MXC_CPU_MX6ULL) || is_cpu_type(MXC_CPU_MX6SLL))
818         return;
819 
820     /* Due to hardware limitation, on MX6Q we need to gate/ungate
821      * all PFDs to make sure PFD is working right, otherwise, PFDs
822      * may not output clock after reset, MX6DL and MX6SL have added
823      * 396M pfd workaround in ROM code, as bus clock need it
824      */
825 
826     mask480 = ANATOP_PFD_CLKGATE_MASK(0) |
827         ANATOP_PFD_CLKGATE_MASK(1) |
828         ANATOP_PFD_CLKGATE_MASK(2) |
829         ANATOP_PFD_CLKGATE_MASK(3);
830     mask528 = ANATOP_PFD_CLKGATE_MASK(1) |
831         ANATOP_PFD_CLKGATE_MASK(3);
832 
833     reg = readl(&ccm->cbcmr);
834     periph2 = ((reg & MXC_CCM_CBCMR_PRE_PERIPH2_CLK_SEL_MASK)
835         >> MXC_CCM_CBCMR_PRE_PERIPH2_CLK_SEL_OFFSET);
836     periph1 = ((reg & MXC_CCM_CBCMR_PRE_PERIPH_CLK_SEL_MASK)
837         >> MXC_CCM_CBCMR_PRE_PERIPH_CLK_SEL_OFFSET);
838 
839   /* Checking if PLL2 PFD0 or PLL2 PFD2 is using for periph clock */
840     if ((periph2 != 0x2) && (periph1 != 0x2))
841         mask528 |= ANATOP_PFD_CLKGATE_MASK(0);
842 
843     if ((periph2 != 0x1) && (periph1 != 0x1) &&
844         (periph2 != 0x3) && (periph1 != 0x3))
845         mask528 |= ANATOP_PFD_CLKGATE_MASK(2);
846 
847     writel(mask480, &anatop->pfd_480_set);
848     writel(mask528, &anatop->pfd_528_set);
849     writel(mask480, &anatop->pfd_480_clr);
```

```
850     writel(mask528, &anatop->pfd_528_clr);
851 }
```

在第 816 行会判断当前 CPU 类型，如果 CPU 为 MX6SX、MX6UL、MX6ULL 或 MX6SLL 中的任意一种，那么就会直接返回，相当于 s_init 函数什么都没做。所以对于 I.MX6UL/I.MX6ULL 来说，s_init 就是个空函数。从 s_init 函数退出以后进入函数 lowlevel_init，但是 lowlevel_init 函数也执行完成了，返回到了函数 cpu_init_crit，函数 cpu_init_crit 也执行完成了，最终返回到 save_boot_params_ret，函数调用路径如图 32.2.3.1 所示：

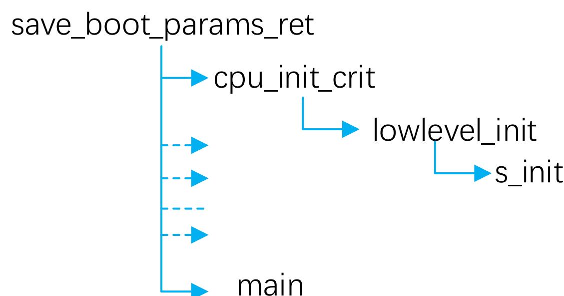


图 32.2.3.1 uboot 函数调用路径

从图 32.2.3.1 可知，接下来要执行的是 save_boot_params_ret 中的_main 函数，接下来分析_main 函数。

32.2.4 _main 函数详解

_main 函数定义在文件 arch/arm/lib/crt0.S 中，函数内容如下：

示例代码 32.2.4.1 crt0.S 代码段

```
63 /*
64  * entry point of crt0 sequence
65  */
66
67 ENTRY(_main)
68
69 /*
70  * Set up initial C runtime environment and call board_init_f(0).
71  */
72
73 #if defined(CONFIG_SPL_BUILD) && defined(CONFIG_SPL_STACK)
74     ldr sp, =(CONFIG_SPL_STACK)
75 #else
76     ldr sp, =(CONFIG_SYS_INIT_SP_ADDR)
77 #endif
78 #if defined(CONFIG_CPU_V7M) /* v7M forbids using SP as BIC
destination */
79     mov r3, sp
80     bic r3, r3, #7
81     mov sp, r3
82 #else
```

```
83      bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
84  #endif
85      mov r0, sp
86      bl  board_init_f_alloc_reserve
87      mov sp, r0
88      /* set up gd here, outside any C code */
89      mov r9, r0
90      bl  board_init_f_init_reserve
91
92      mov r0, #0
93      bl  board_init_f
94
95  #if ! defined(CONFIG_SPL_BUILD)
96
97  /*
98   * Set up intermediate environment (new sp and gd) and call
99   * relocate_code(addr_moni). Trick here is that we'll return
100  * 'here' but relocated.
101  */
102
103     ldr sp, [r9, #GD_START_ADDR_SP] /* sp = gd->start_addr_sp */
104 #if defined(CONFIG_CPU_V7M) /* v7M forbids using SP as BIC
destination */
105     mov r3, sp
106     bic r3, r3, #7
107     mov sp, r3
108 #else
109     bic sp, sp, #7  /* 8-byte alignment for ABI compliance */
110 #endif
111     ldr r9, [r9, #GD_BD]        /* r9 = gd->bd */
112     sub r9, r9, #GD_SIZE        /* new GD is below bd */
113
114     adr lr, here
115     ldr r0, [r9, #GD_RELOC_OFF]     /* r0 = gd->reloc_off */
116     add lr, lr, r0
117 #if defined(CONFIG_CPU_V7M)
118     orr lr, #1              /* As required by Thumb-only */
119 #endif
120     ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr */
121     b   relocate_code
122 here:
123 /*
124  * now relocate vectors
```

```
125  */
126 
127     bl  relocate_vectors
128 
129 /* Set up final (full) environment */
130 
131     bl  c_runtime_cpu_setup /* we still call old routine here */
132 #endif
133 #if !defined(CONFIG_SPL_BUILD) || defined(CONFIG_SPL_FRAMEWORK)
134 # ifdef CONFIG_SPL_BUILD
135     /* Use a DRAM stack for the rest of SPL, if requested */
136     bl  spl_relocate_stack_gd
137     cmp r0, #0
138     movne   sp, r0
139     movne   r9, r0
140 # endif
141     ldr r0, =__bss_start    /* this is auto-relocated! */
142 
143 #ifdef CONFIG_USE_ARCH_MEMSET
144     ldr r3, =__bss_end      /* this is auto-relocated! */
145     mov r1, #0x00000000     /* prepare zero to clear BSS */
146 
147     subs    r2, r3, r0      /* r2 = memset len */
148     bl  memset
149 #else
150     ldr r1, =__bss_end      /* this is auto-relocated! */
151     mov r2, #0x00000000     /* prepare zero to clear BSS */
152 
153 clbss_l:cmp r0, r1          /* while not at end of BSS */
154 #if defined(CONFIG_CPU_V7M)
155     itt lo
156 #endif
157     strlo   r2, [r0]        /* clear 32-bit BSS word */
158     addlo   r0, r0, #4      /* move to next */
159     blo clbss_l
160 #endif
161 
162 #if ! defined(CONFIG_SPL_BUILD)
163     bl coloured_LED_init
164     bl red_led_on
165 #endif
166     /* call board_init_r(gd_t *id, ulong dest_addr) */
167     mov     r0, r9                  /* gd_t */
```

```
168     ldr r1, [r9, #GD_RELOCADDR] /* dest_addr */
169     /* call board_init_r */
170 #if defined(CONFIG_SYS_THUMB_BUILD)
171     ldr lr, =board_init_r   /* this is auto-relocated! */
172     bx  lr
173 #else
174     ldr pc, =board_init_r   /* this is auto-relocated! */
175 #endif
176     /* we should not return here. */
177 #endif
178
179 ENDPROC(_main)
```

第 76 行，设置 sp 指针为 CONFIG_SYS_INIT_SP_ADDR，也就是 sp 指向 0X0091FF00。

第 83 行，sp 做 8 字节对齐。

第 85 行，读取 sp 到寄存器 r0 里面，此时 r0=0X0091FF00。

第 86 行，调用函数 board_init_f_alloc_reserve，此函数有一个参数，参数为 r0 中的值，也就是 0X0091FF00，此函数定义在文件 common/init/board_init.c 中，内容如下：

示例代码 32.2.4.2 board_init.c 代码段

```
56 ulong board_init_f_alloc_reserve(ulong top)
57 {
58     /* Reserve early malloc arena */
59     #if defined(CONFIG_SYS_MALLOC_F)
60         top -= CONFIG_SYS_MALLOC_F_LEN;
61     #endif
62     /* LAST : reserve GD (rounded up to a multiple of 16 bytes) */
63     top = rounddown(top-sizeof(struct global_data), 16);
64
65     return top;
66 }
```

函数 board_init_f_alloc_reserve 主要是留出早期的 malloc 内存区域和 gd 内存区域，其中 CONFIG_SYS_MALLOC_F_LEN=0X400(在文件 include/generated/autoconf.h 中定义)，sizeof(struct global_data)=248(GD_SIZE 值)，完成以后的内存分布如图 32.2.4.1 所示：

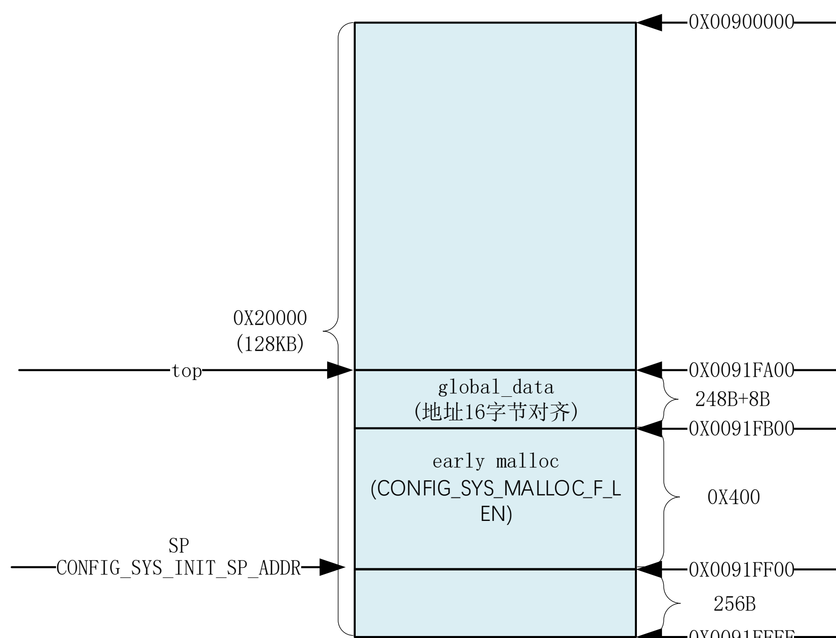


图 32.2.4.1 内存分布图

函数 board_init_f_alloc_reserve 是有返回值的，返回值为新的 top 值，从图 32.2.4.1 可知，此时 top=0X0091FA00。

继续回到示例代码 32.2.4.1 中，第 87 行，将 r0 写入到 sp 里面，r0 保存着函数 board_init_f_alloc_reserve 的返回值，所以这一句也就是设置 sp=0X0091FA00。

第 89 行，将 r0 寄存器的值写到寄存器 r9 里面，因为 r9 寄存器存放着全局变量 gd 的地址，在文件 arch/arm/include/asm/global_data.h 中有如图 32.2.4.2 所示宏定义：

```
83  #ifdef CONFIG_ARM64
84  #define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("x18")
85  #else
86  #define DECLARE_GLOBAL_DATA_PTR     register volatile gd_t *gd asm ("r9")
87  #endif
88  #endif
```

图 32.2.4.2 DECLARE_GLOBAL_DATA_PTR 宏定义

从图 32.2.4.2 可以看出，uboot 中定义了一个指向 gd_t 的指针 gd，gd 存放在寄存器 r9 里面的，因此 gd 是个全局变量。gd_t 是个结构体，在 include/asm-generic/global_data.h 里面有定义，gd_定义如下：

示例代码 32.2.4.3 global_data.h 代码段

```
27 typedef struct global_data {
28     bd_t *bd;
29     unsigned long flags;
30     unsigned int baudrate;
31     unsigned long cpu_clk;  /* CPU clock in Hz!     */
32     unsigned long bus_clk;
33     /* We cannot bracket this with CONFIG_PCI due to mpc5xxx */
34     unsigned long pci_clk;
35     unsigned long mem_clk;
```

```
36 #if defined(CONFIG_LCD) || defined(CONFIG_VIDEO)
37     unsigned long fb_base;  /* Base address of framebuffer mem */
38 #endif
......
121 #ifdef CONFIG_DM_VIDEO
122     ulong video_top;        /* Top of video frame buffer area */
123     ulong video_bottom;     /* Bottom of video frame buffer area */
124 #endif
125 } gd_t;
```

因此这一行代码就是设置 gd 所指向的位置，也就是 gd 指向 0X0091FA00。

继续回到示例代码 32.2.4.1 中，第 90 行调用函数 board_init_f_init_reserve，此函数在文件 common/init/board_init.c 中有定义，函数内容如下：

示例代码 32.2.4.4 board_init.c 代码段

```
110 void board_init_f_init_reserve(ulong base)
111 {
112     struct global_data *gd_ptr;
113 #ifndef _USE_MEMCPY
114     int *ptr;
115 #endif
116 
117     /*
118      * clear GD entirely and set it up.
119      * Use gd_ptr, as gd may not be properly set yet.
120      */
121 
122     gd_ptr = (struct global_data *)base;
123     /* zero the area */
124 #ifdef _USE_MEMCPY
125     memset(gd_ptr, '\0', sizeof(*gd));
126 #else
127     for (ptr = (int *)gd_ptr; ptr < (int *)(gd_ptr + 1); )
128         *ptr++ = 0;
129 #endif
130     /* set GD unless architecture did it already */
131 #if !defined(CONFIG_ARM)
132     arch_setup_gd(gd_ptr);
133 #endif
134     /* next alloc will be higher by one GD plus 16-byte alignment */
135     base += roundup(sizeof(struct global_data), 16);
136 
137     /*
138      * record early malloc arena start.
139      * Use gd as it is now properly set for all architectures.
```

```
140     */
141
142 #if defined(CONFIG_SYS_MALLOC_F)
143    /* go down one 'early malloc arena' */
144    gd->malloc_base = base;
145    /* next alloc will be higher by one 'early malloc arena' size */
146    base += CONFIG_SYS_MALLOC_F_LEN;
147 #endif
148 }
```

可以看出，此函数用于初始化 gd，其实就是清零处理。另外，此函数还设置了 gd->malloc_base 为 gd 基地址+gd 大小=0X0091FA00+248=0X0091FAF8，在做 16 字节对齐，最终 gd->malloc_base=0X0091FB00，这个也就是 early malloc 的起始地址。

继续回到示例代码 32.2.4.1 中，第 92 行设置 R0 为 0。

第 93 行，调用 board_init_f 函数，此函数定义在文件 common/board_f.c 中！主要用来初始化 DDR，定时器，完成代码拷贝等等，此函数我们后面在详细的分析。

第 103 行，重新设置环境(sp 和 gd)、获取 gd->start_addr_sp 的值赋给 sp，在函数 board_init_f 中会初始化 gd 的所有成员变量，其中 gd->start_addr_sp=0X9EF44E90， 所以这里相当于设置 sp=gd->start_addr_sp=0X9EF44E90。0X9EF44E90 是 DDR 中的地址，说明新的 sp 和 gd 将会存放到 DDR 中，而不是内部的 RAM 了。GD_START_ADDR_SP=64，参考示例代码 32.2.2.4。

第 109 行，sp 做 8 字节对齐。

第 111 行，获取 gd->bd 的地址赋给 r9，此时 r9 存放的是老的 gd，这里通过获取 gd->bd 的地址来计算出新的 gd 的位置。GD_BD=0，参考示例代码 32.2.2.4。

第 112 行，新的 gd 在 bd 下面，所以 r9 减去 gd 的大小就是新的 gd 的位置，获取到新的 gd 的位置以后赋值给 r9。

第 114 行，设置 lr 寄存器为 here，这样后面执行其他函数返回的时候就返回到了第 122 行的 here 位置处。

第 115，读取 gd->reloc_off 的值复制给 r0 寄存器，GD_RELOC_OFF=68，参考示例代码 32.2.2.4。

第 116 行，lr 寄存器的值加上 r0 寄存器的值，重新赋值给 lr 寄存器。因为接下来要重定位代码，也就是把代码拷贝到新的地方去(现在的 uboot 存放的起始地址为 0X87800000，下面要将 uboot 拷贝到 DDR 最后面的地址空间出，将 0X87800000 开始的内存空出来)，其中就包括 here，因此 lr 中的 here 要使用重定位后的位置。

第 120 行，读取 gd->relocaddr 的值赋给 r0 寄存器，此时 r0 寄存器就保存着 uboot 要拷贝的目的地址，为 0X9FF47000。GD_RELOCADDR=48，参考示例代码 32.2.2.4。

第 121 行，调用函数 relocate_code，也就是代码重定位函数，此函数负责将 uboot 拷贝到新的地方去，此函数定义在文件 arch/arm/lib/relocate.S 中稍后会详细分析此函数。

第 127 行，调用函数 relocate_vectors，对中断向量表做重定位，此函数定义在文件 arch/arm/lib/relocate.S 中，稍后会详细分析此函数。

第 131 行，调用函数 c_runtime_cpu_setup，此函数定义在文件 arch/arm/cpu/armv7/start.S 中，函数内容如下：

示例代码 32.2.4.5 start.S 代码段

```
77 ENTRY(c_runtime_cpu_setup)
78 /*
```

```
79  * If I-cache is enabled invalidate it
80  */
81 #ifndef CONFIG_SYS_ICACHE_OFF
82     mcr p15, 0, r0, c7, c5, 0   @ invalidate icache
83     mcr    p15, 0, r0, c7, c10, 4  @ DSB
84     mcr    p15, 0, r0, c7, c5, 4   @ ISB
85 #endif
86
87     bx  lr
88
89 ENDPROC(c_runtime_cpu_setup)
```

第 141~159 行，清除 BSS 段。

第 167 行，设置函数 board_init_r 的两个参数，函数 board_init_r 声明如下：

```
board_init_r(gd_t *id, ulong dest_addr)
```

第一个参数是 gd，因此读取 r9 保存到 r0 里面。

第 168 行，设置函数 board_init_r 的第二个参数是目的地址，因此 r1= gd->relocaddr。

第 174 行、调用函数 board_init_r，此函数定义在文件 common/board_r.c 中，稍后会详细的分析此函数。

这个就是_main 函数的运行流程，在_main 函数里面调用了 board_init_f、relocate_code、relocate_vectors 和 board_init_r 这 4 个函数，接下来依次看一下这 4 个函数都是干啥的。

32.2.5 board_init_f 函数详解

_main 中会 board_init_f 函数，board_init_f 函数主要有两个工作：

①、初始化一系列外设，比如串口、定时器，或者打印一些消息等。

②、初始化 gd 的各个成员变量，uboot 会将自己重定位到 DRAM 最后面的地址区域，也就是将自己拷贝到 DRAM 最后面的内存区域中。这么做的目的是给 Linux 腾出空间，防止 Linux kernel 覆盖掉 uboot，将 DRAM 前面的区域完整的空出来。在拷贝之前肯定要给 uboot 各部分分配好内存位置和大小，比如 gd 应该存放到哪个位置，malloc 内存池应该存放到哪个位置等等。这些信息都保存在 gd 的成员变量中，因此要对 gd 的这些成员变量做初始化。最终形成一个完整的内存“分配图”，在后面重定位 uboot 的时候就会用到这个内存“分配图”。

此函数定义在文件 common/board_f.c 中定义，代码如下：

示例代码 32.2.5.1 board_f.c 代码段

```
1035 void board_init_f(ulong boot_flags)
1036 {
1037 #ifdef CONFIG_SYS_GENERIC_GLOBAL_DATA
1038   /*
1039    * For some archtectures, global data is initialized and used
1040    * before  calling this function. The data should be preserved.
1041    * For others, CONFIG_SYS_GENERIC_GLOBAL_DATA should be defined
1042    * and use the stack here to host global data until relocation.
1043    */
1044   gd_t data;
1045
```

```
1046    gd = &data;
1047
1048    /*
1049     * Clear global data before it is accessed at debug print
1050     * in initcall_run_list. Otherwise the debug print probably
1051     * get the wrong vaule of gd->have_console.
1052     */
1053    zero_global_data();
1054 #endif
1055
1056    gd->flags = boot_flags;
1057    gd->have_console = 0;
1058
1059    if (initcall_run_list(init_sequence_f))
1060        hang();
1061
1062 #if !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX) && \
1063        !defined(CONFIG_EFI_APP)
1064    /* NOTREACHED - jump_to_copy() does not return */
1065    hang();
1066 #endif
1067 }
```

因为没有定义 CONFIG_SYS_GENERIC_GLOBAL_DATA，所以第 1037~1054 行代码无效。

第 1056 行，初始化 gd->flags=boot_flags=0。

第 1057 行，设置 gd->have_console=0。

重点在第 1059 行！通过函数 initcall_run_list 来运行初始化序列 init_sequence_f 里面的一些列函数，init_sequence_f 里面包含了一系列的初始化函数，init_sequence_f 也是定义在文件 common/board_f.c 中，由于 init_sequence_f 的内容比较长，里面有大量的条件编译代码，这里为了缩小篇幅，将条件编译部分删除掉了，去掉条件编译以后的 init_sequence_f 定义如下：

示例代码 32.2.5.1 board_f.c 代码段

```
/*****************去掉条件编译语句后的 init_sequence_f***************/
1  static init_fnc_t init_sequence_f[] = {
2       setup_mon_len,
3       initf_malloc,
4       initf_console_record,
5       arch_cpu_init,        /* basic arch cpu dependent setup  */
6       initf_dm,
7       arch_cpu_init_dm,
8       mark_bootstage,       /* need timer, go after init dm    */
9       board_early_init_f,
10      timer_init,           /* initialize timer                */
11      board_postclk_init,
12      get_clocks,
```

```
13      env_init,             /* initialize environment          */
14      init_baud_rate,       /* initialze baudrate settings     */
15      serial_init,          /* serial communications setup     */
16      console_init_f,       /* stage 1 init of console         */
17      display_options,      /* say that we are here            */
18      display_text_info,    /* show debugging info if required */
19      print_cpuinfo,        /* display cpu info (and speed)    */
20      show_board_info,
21      INIT_FUNC_WATCHDOG_INIT
22      INIT_FUNC_WATCHDOG_RESET
23      init_func_i2c,
24      announce_dram_init,
25      /* TODO: unify all these dram functions? */
26      dram_init,                /* configure available RAM banks */
27      post_init_f,
28      INIT_FUNC_WATCHDOG_RESET
29      testdram,
30      INIT_FUNC_WATCHDOG_RESET
31      INIT_FUNC_WATCHDOG_RESET
32      /*
33      * Now that we have DRAM mapped and working, we can
34      * relocate the code and continue running from DRAM.
35      *
36      * Reserve memory at end of RAM for (top down in that order):
37      *  - area that won't get touched by U-Boot and Linux (optional)
38      *  - kernel log buffer
39      *  - protected RAM
40      *  - LCD framebuffer
41      *  - monitor code
42      *  - board info struct
43      */
44      setup_dest_addr,
45      reserve_round_4k,
46      reserve_mmu,
47      reserve_trace,
48      reserve_uboot,
49      reserve_malloc,
50      reserve_board,
51      setup_machine,
52      reserve_global_data,
53      reserve_fdt,
54      reserve_arch,
55      reserve_stacks,
```

```
56      setup_dram_config,
57      show_dram_config,
58      display_new_sp,
59      INIT_FUNC_WATCHDOG_RESET
60      reloc_fdt,
61      setup_reloc,
62      NULL,
63 };
```

接下来分析以上函数执行完以后的结果：

第 2 行，setup_mon_len 函数设置 gd 的 mon_len 成员变量，此处为__bss_end -_start，也就是整个代码的长度。0X878A8E74-0x87800000=0XA8E74，这个就是代码长度

第 3 行，initf_malloc 函数初始化 gd 中跟 malloc 有关的成员变量，比如 malloc_limit，此函数会设置 gd->malloc_limit＝CONFIG_SYS_MALLOC_F_LEN=0X400。malloc_limit 表示 malloc 内存池大小。

第 4 行，initf_console_record，如果定义了宏 CONFIG_CONSOLE_RECORD 和宏 CONFIG_SYS_MALLOC_F_LEN 的话此函数就会调用函数 console_record_init，但是 IMX6ULL 的 uboot 没有定义宏 CONFIG_CONSOLE_RECORD，所以此函数直接返回 0。

第 5 行，arch_cpu_init 函数，初始化架构相关的内容，CPU 级别的操作。

第 6 行，initf_dm 函数，驱动模型的一些初始化。

第 7 行，arch_cpu_init_dm 函数未实现。

第 8 行，mark_bootstage 函数应该是和啥标记有关的

第 9 行，board_early_init_f 函数，板子相关的早期的一些初始化设置，I.MX6ULL 用来初始化串口的 IO 配置

第 10 行，timer_init，初始化定时器，Cortex-A7 内核有一个定时器，这里初始化的就是 Cortex-A 内核的那个定时器。通过这个定时器来为 uboot 提供时间。就跟 Cortex-M 内核 Systick 定时器一样。关于 Cortex-A 内部定时器的详细内容，请参考文档《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》的“Chapter B8 The Generic Timer”章节。

第 11 行，board_postclk_init，对于 I.MX6ULL 来说是设置 VDDSOC 电压。

第 12 行，get_clocks 函数用于获取一些时钟值，I.MX6ULL 获取的是 sdhc_clk 时钟，也就是 SD 卡外设的时钟。

第 13 行，env_init 函数是和环境变量有关的，设置 gd 的成员变量 env_addr，也就是环境变量的保存地址。

第 14 行，init_baud_rate 函数用于初始化波特率，根据环境变量 baudrate 来初始化 gd->baudrate。

第 15 行，serial_init，初始化串口。

第 16 行，console_init_f，设置 gd->have_console 为 1，表示有个控制台，此函数也将前面暂存在缓冲区中的数据通过控制台打印出来。

第 17 行、display_options，通过串口输出一些信息，如图 32.2.5.1 所示：

```
U-Boot 2016.03 (Jul 14 2018 - 17:08:43 +0800)
```

图 32.2.5.1 串口信息输出

第 18 行，display_text_info，打印一些文本信息，如果开启 UBOOT 的 DEBUG 功能的话就会输出 text_base、bss_start、bss_end，形式如下：

```
debug("U-Boot code: %08lX -> %08lX    BSS: -> %08lX\n",text_base, bss_start, bss_end);
```

结果如图 32.2.5.2 所示：

```
U-Boot 2016.03 (Aug 01 2018 - 09:44:06 +0800)

initcall: 878119cc
U-Boot code: 87800000 -> 878665E0  BSS: -> 878B1EF8
initcall: 878028ac
CPU:   Freescale i.MX6ULL rev1.0 528 MHz (running at 396 MHz)
CPU:   Commercial temperature grade (0C to 95C)malloc_simple: size=10, pt
uclass_find_device_by_seq: 0 -1
uclass_find_device_by_seq: 0 0
```

图 32.2.5.2 文本信息

第 19 行，print_cpuinfo 函数用于打印 CPU 信息，结果如图 32.2.5.3 所示：

```
CPU:   Freescale i.MX6ULL rev1.0 528 MHz (running at 396 MHz)
CPU:   Commercial temperature grade (0C to 95C) at 47C
Reset cause: WDOG
```

图 32.2.5.3 CPU 信息

第 20 行，show_board_info 函数用于打印板子信息，会调用 checkboard 函数，结果如图 32.2.5.4 所示：

```
CPU:   Freescale i.MX6ULL rev1.0 528 MHz (running at 396 MHz
CPU:   Commercial temperature grade (0C to 95C) at 42C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
```

图 32.2.5.4 板子信息

第 21 行，INIT_FUNC_WATCHDOG_INIT，初始化看门狗，对于 I.MX6ULL 来说是空函数

第 22 行，INIT_FUNC_WATCHDOG_RESET，复位看门狗，对于 I.MX6ULL 来说是空函数

第 23 行，init_func_i2c 函数用于初始化 I2C，初始化完成以后会输出如图 32.2.5.5 所示信息：

```
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT43AB (480x272)
Video: 480x272x24
In:    serial
```

图 32.2.5.5 I2C 初始化信息输出

第 24 行，announce_dram_init，此函数很简单，就是输出字符串“DRAM:”

第 26 行，dram_init，并非真正的初始化 DDR，只是设置 gd->ram_size 的值，对于正点原子 I.MX6ULL 开发板 EMMC 版本核心板来说就是 512MB。

第 27 行，post_init_f，此函数用来完成一些测试，初始化 gd->post_init_f_time

第 29 行，testdram，测试 DRAM，空函数。

第 44 行，setup_dest_addr 函数，设置目的地址，设置 gd->ram_size，gd->ram_top，gd->relocaddr 这三个的值。接下来我们会遇到很多跟数值有关的设置，如果直接看代码分析的话就太费时间了，我可以修改 uboot 代码，直接将这些值通过串口打印出来，比如这里我们修改文件 common/board_f.c，因为 setup_dest_addr 函数定义在文件 common/board_f.c 中，在 setup_dest_addr 函数输入如图 32.2.5.6 所示内容：

```
358	gd->ram_top += get_effective_memsize();
359	gd->ram_top = board_get_usable_ram_top(gd->mon_len);
360	gd->relocaddr = gd->ram_top;
361	debug("Ram top: %08lX\n", (ulong)gd->ram_top);
362 #if defined(CONFIG_MP) && (defined(CONFIG_MPC86xx) || defined(CONFIG_E500))
363	/*
364	 * We need to make sure the location we intend to put secondary core
365	 * boot code is reserved and not used by any part of u-boot
366	 */
367	if (gd->relocaddr > determine_mp_bootpg(NULL)) {
368		gd->relocaddr = determine_mp_bootpg(NULL);
369		debug("Reserving MP boot page to %08lx\n", gd->relocaddr);
370	}
371 #endif
372
373	printf("gd->ram_size %#x\r\n", gd->ram_size);
374	printf("gd->ram_top %#x\r\n", gd->ram_top);
375	printf("gd->relocaddr %#x\r\n",gd->relocaddr);
376
377	return 0;
378 }
```

通过串口输出这三个成员变量的值

图 32.2.5.6 添加 print 函数打印成员变量值

设置好以后重新编译 uboot，然后烧写到 SD 卡中，选择 SD 卡启动，重启开发板，打开 SecureCRT，uboot 会输出如图 32.2.5.7 所示信息：

```
DRAM:  gd->ram_size 0x20000000
gd->ram_top 0xa0000000
gd->relocaddr 0xa0000000
```

图 32.2.5.7 信息输出

从图 32.2.5.7 可以看出：

```
gd->ram_size = 0X20000000      //ram 大小为 0X20000000=512MB
gd->ram_top = 0XA0000000       //ram 最高地址为 0X80000000+0X20000000=0XA0000000
gd->relocaddr = 0XA0000000     //重定位后最高地址为 0XA0000000
```

第 45 行，reserve_round_4k 函数用于对 gd->relocaddr 做 4KB 对齐，因为 gd->relocaddr=0XA0000000，已经是 4K 对齐了，所以调整后不变。

第 46 行，reserve_mmu，留出 MMU 的 TLB 表的位置，分配 MMU 的 TLB 表内存以后会对 gd->relocaddr 做 64K 字节对齐。完成以后 gd->arch.tlb_size、gd->arch.tlb_addr 和 gd->relocaddr 如图 32.2.5.8 所示：

```
gd->arch.tlb_size 0x4000
gd->arch.tlb_add 0x9fff0000
gd->relocaddr 0x9fff0000
```

图 32.2.5.8 信息输出

从图 32.2.5.8 可以看出：

```
gd->arch.tlb_size= 0X4000              //MMU 的 TLB 表大小
gd->arch.tlb_addr=0X9FFF0000           //MMU 的 TLB 表起始地址，64KB 对齐以后
gd->relocaddr=0X9FFF0000               //relocaddr 地址
```

第 47 行，reserve_trace 函数，留出跟踪调试的内存，I.MX6ULL 没有用到！

第 48 行，reserve_uboot，留出重定位后的 uboot 所占用的内存区域，uboot 所占用大小由 gd->mon_len 所指定，留出 uboot 的空间以后还要对 gd->relocaddr 做 4K 字节对齐，并且重新设置 gd->start_addr_sp，结果如图 32.2.5.9 所示：

```
gd->mon_len =0XA8EF4
gd->start_addr_sp  =0X9FF47000
gd->relocaddr  =0X9FF47000
```

图 32.2.5.9 信息输出

从图 32.2.5.9 可以看出：

```
gd->mon_len =  0XA8EF4
gd->start_addr_sp = 0X9FF47000
gd->relocaddr = 0X9FF47000
```

第 49 行，reserve_malloc，留出 malloc 区域，调整 gd->start_addr_sp 位置，malloc 区域由宏 TOTAL_MALLOC_LEN 定义，宏定义如下：

```
#define TOTAL_MALLOC_LEN  (CONFIG_SYS_MALLOC_LEN +
CONFIG_ENV_SIZE)
```

mx6ull_alientek_emmc.h 文件中定义宏 CONFIG_SYS_MALLOC_LEN 为 16MB=0X1000000，宏 CONFIG_ENV_SIZE=8KB=0X2000，因此 TOTAL_MALLOC_LEN=0X1002000。调整以后 gd->start_addr_sp 如图 32.2.5.10 所示：

```
TOTAL_MALLOC_LEN =0X1002000
gd->start_addr_sp  =0X9EF45000
```

图 32.2.5.10 信息输出

从图 32.2.5.10 可以看出：

```
TOTAL_MALLOC_LEN=0X1002000
gd->start_addr_sp=0X9EF45000    //0X9FF47000-16MB-8KB=0X9EF45000
```

第 50 行，reserve_board 函数，留出板子 bd 所占的内存区，bd 是结构体 bd_t，bd_t 大小为 80 字节，结果如图 32.2.5.11 所示：

```
gd->bd = 0X9EF44FB0
gd->start_addr_sp = 0X9EF44FB0
```

图 32.2.5.10 信息输出

从图 32.2.5.11 可以看出：

```
gd->start_addr_sp=0X9EF44FB0
gd->bd=0X9EF44FB0
```

第 51 行，setup_machine，设置机器 ID，linux 启动的时候会和这个机器 ID 匹配，如果匹配的话 linux 就会启动正常。但是!! I.MX6ULL 不用这种方式了，这是以前老版本的 uboot 和 linux 使用的，新版本使用设备树了，因此此函数无效。

第 52 行，reserve_global_data 函数，保留出 gd_t 的内存区域，gd_t 结构体大小为 248B，结果如图 32.2.5.11 所示：

```
gd->new_gd = 0x9ef44eb8
gd->start_addr_sp = 0x9ef44eb8
```

图 32.2.5.11 信息输出

```
gd->start_addr_sp=0X9EF44EB8    //0X9EF44FB0-248=0X9EF44EB8
gd->new_gd=0X9EF44EB8
```

第 53 行，reserve_fdt，留出设备树相关的内存区域，I.MX6ULL 的 uboot 没有用到，因此此函数无效。

第 54 行，reserve_arch 是个空函数。

第 55 行，reserve_stacks，留出栈空间，先对 gd->start_addr_sp 减去 16，然后做 16 字节对

其。如果使能 IRQ 的话还要留出 IRQ 相应的内存，具体工作是由 arch/arm/lib/stack.c 文件中的函数 arch_reserve_stacks 完成。结果如图 32.2.5.12 所示：

```
gd->start_addr_sp = 0x9ef44e90
```

图 32.2.5.12 信息输出

在本 uboot 中并没有使用到 IRQ，所以不会留出 IRQ 相应的内存区域，此时：

```
gd->start_addr_sp=0X9EF44E90
```

第 56 行，setup_dram_config 函数设置 dram 信息，就是设置 gd->bd->bi_dram[0].start 和 gd->bd->bi_dram[0].size，后面会传递给 linux 内核，告诉 linux DRAM 的起始地址和大小。结果如图 32.2.5.13 所示：

```
gd->bd->bi_dram[0].start = 0x80000000
gd->bd->bi_dram[0].size = 0x20000000
```

图 32.2.5.13 信息输出

从图 32.2.5.13 可以看出，DRAM 的起始地址为 0X80000000，大小为 0X20000000(512MB)。

第 57 行，show_dram_config 函数，用于显示 DRAM 的配置，如图 32.2.5.14 所示：

```
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
```

图 32.2.5.14 信息输出

第 58 行，display_new_sp 函数，显示新的 sp 位置，也就是 gd->start_addr_sp，不过要定义宏 DEBUG，结果如图 32.2.5.15 所示：

```
Reset cause: unknown reset
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
New Stack Pointer is: 9ef44e90
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT43AB (480x272)
Video: 480x272x24
In:    serial
```

图 32.2.5.15 信息输出

图 32.2.5.15 中的 gd->start_addr_sp 值和我们前面分析的最后一次修改的值一致。

第 60 行，reloc_fdt 函数用于重定位 fdt，没有用到。

第 61 行，setup_reloc，设置 gd 的其他一些成员变量，供后面重定位的时候使用，并且将以前的 gd 拷贝到 gd->new_gd 处。需要使能 DEBUG 才能看到相应的信息输出，如图 32.2.5.16 所示：

```
DRAM:  512 MiB
Relocation Offset is: 18747000
Relocating to 9ff47000, new gd at 9ef44eb8, sp at 9ef44e90
```

图 32.2.5.16 信息输出

从图 32.2.5.16 可以看出，uboot 重定位后的偏移为 0X18747000，重定位后的新地址为 0X9FF4700，新的 gd 首地址为 0X9EF44EB8，最终的 sp 为 0X9EF44E90。

至此，board_init_f 函数就执行完成了，最终的内存分配如图 32.2.5.16 所示：

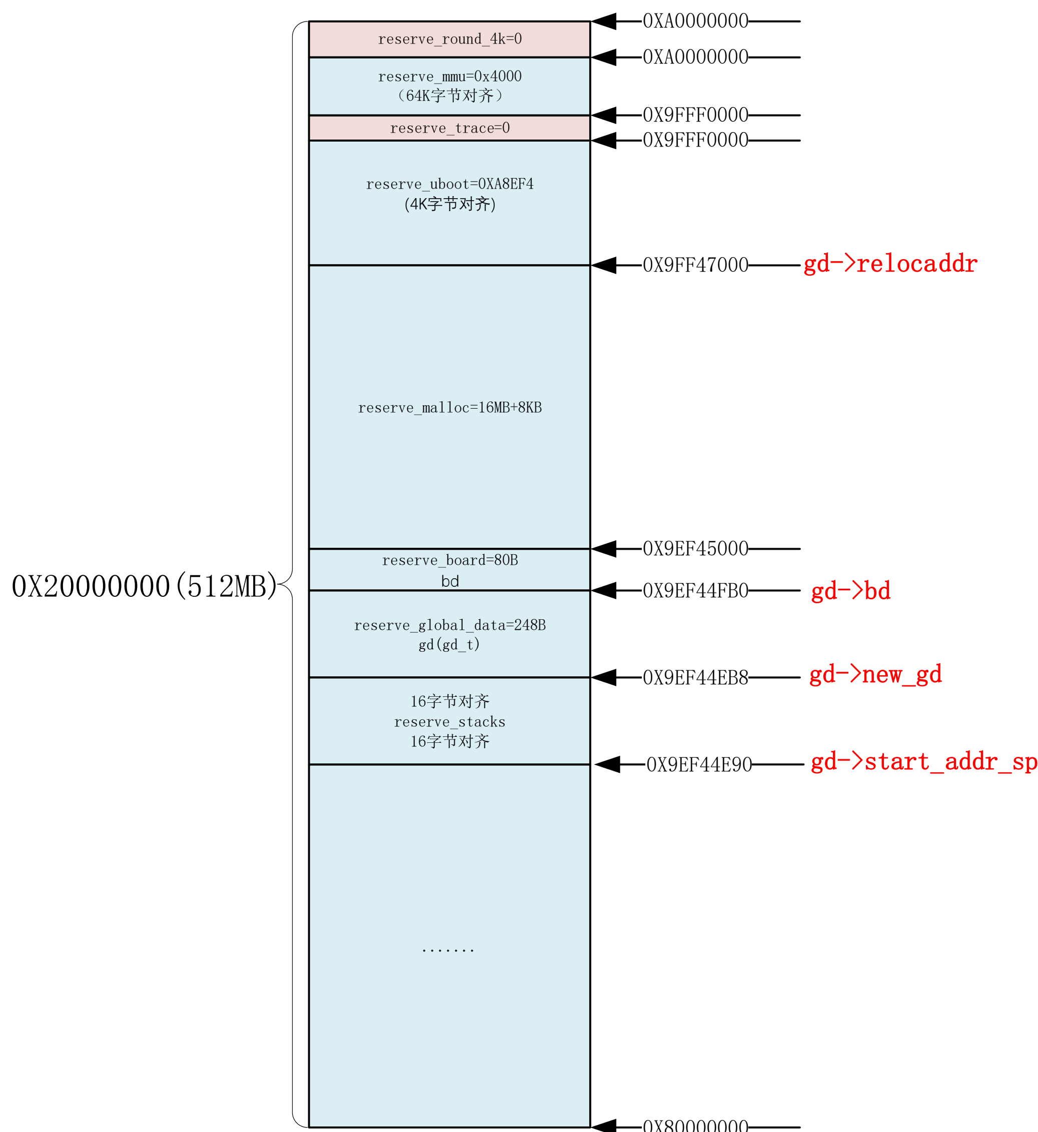


图 32.2.5.16 最终的内存分配图

32.2.6 relocate_code 函数详解

relocate_code 函数是用于代码拷贝的，此函数定义在文件 arch/arm/lib/relocate.S 中，代码如下：

示例代码 32.2.6.1 relocate.S 代码段

```
/*
 * void relocate_code(addr_moni)
 *
 * This function relocates the monitor code.
 *
 * NOTE:
 * To prevent the code below from containing references with an
 * R_ARM_ABS32 relocation record type, we never refer to linker-
```

```
 * defined symbols directly. Instead, we declare literals which
 * contain their relative location with respect to relocate_code,
 * and at run time, add relocate_code back to them.
  */

79 ENTRY(relocate_code)
80     ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
81     subs    r4, r0, r1          /* r4 <- relocation offset */
82     beq relocate_done           /* skip relocation */
83     ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
84 
85 copy_loop:
86     ldmia   r1!, {r10-r11}   /* copy from source address [r1]    */
87     stmia   r0!, {r10-r11}   /* copy to   target address [r0]    */
88     cmp r1, r2               /* until source end address [r2]    */
89     blo copy_loop
90 
91     /*
92      * fix .rel.dyn relocations
93      */
94     ldr r2, =__rel_dyn_start /* r2 <- SRC &__rel_dyn_start */
95     ldr r3, =__rel_dyn_end   /* r3 <- SRC &__rel_dyn_end */
96 fixloop:
97     ldmia   r2!, {r0-r1}     /* (r0,r1) <- (SRC location,fixup) */
98     and r1, r1, #0xff
99     cmp r1, #23              /* relative fixup? */
100    bne fixnext
101
102    /* relative fix: increase location by offset */
103    add r0, r0, r4
104    ldr r1, [r0]
105    add r1, r1, r4
106    str r1, [r0]
107 fixnext:
108    cmp r2, r3
109    blo fixloop
110
111 relocate_done:
112
113 #ifdef __XSCALE__
114    /*
115     * On xscale, icache must be invalidated and write buffers
116     * drained, even with cache disabled - 4.2.7 of xscale core
```

```
117      developer's manual  */
118     mcr p15, 0, r0, c7, c7, 0   /* invalidate icache */
119     mcr p15, 0, r0, c7, c10, 4  /* drain write buffer */
120 #endif
121
122    /* ARMv4- don't know bx lr but the assembler fails to see that */
123
124 #ifdef __ARM_ARCH_4__
125     mov pc, lr
126 #else
127     bx  lr
128 #endif
129
130 ENDPROC(relocate_code)
```

第 80 行，r1=__image_copy_start，也就是 r1 寄存器保存源地址，由表 31.4.1.1 可知，__image_copy_start=0X87800000。

第 81 行，r0=0X9FF47000，这个地址就是 uboot 拷贝的目标首地址。r4=r0-r1=0X9FF47000-0X87800000=0X18747000，因此 r4 保存偏移量。

第 82 行，如果在第 81 中，r0-r1 等于 0，说明 r0 和 r1 相等，也就是源地址和目的地址是一样的，那肯定就不需要拷贝了！执行 relocate_done 函数

第 83 行，r2=__image_copy_end，r2 中保存拷贝之前的代码结束地址，由表 31.4.1.1 可知，__image_copy_end =0x8785dd54。

第 84 行，函数 copy_loop 完成代码拷贝工作！从 r1，也就是__image_copy_start 开始，读取 uboot 代码保存到 r10 和 r11 中，一次就只拷贝这 2 个 32 位的数据。拷贝完成以后 r1 的值会更新，保存下一个要拷贝的数据地址。

第 87 行，将 r10 和 r11 的数据写到 r0 开始的地方，也就是目的地址。写完以后 r0 的值会更新，更新为下一个要写入的数据地址。

第 88 行，比较 r1 是否和 r2 相等，也就是检查是否拷贝完成，如果不相等的话说明没有拷贝完成，没有拷贝完成的话就跳转到 copy_loop 接着拷贝，直至拷贝完成。

接下来的第 94 行~109 行是重定位.rel.dyn 段，.rel.dyn 段是存放.text 段中需要重定位地址的集合。重定位就是 uboot 将自身拷贝到 DRAM 的另一个地放去继续运行(DRAM 的高地址处)。我们知道，一个可执行的 bin 文件，其链接地址和运行地址要相等，也就是链接到哪个地址，在运行之前就要拷贝到哪个地址去。现在我们重定位以后，运行地址就和链接地址不同了，这样寻址的时候不会出问题吗？为了分析这个问题，我们需要在 mx6ull_alientek_emmc.c 中输入如下所示内容：

示例代码 32.2.6.2 mx6ull_alientek_emmc.c 新添代码段

```
1 static int rel_a = 0;
2
3 void rel_test(void)
4 {
5   rel_a = 100;
6   printf("rel_test\r\n");
7 }
```

最后还需要在 mx6ullevk.c 文件中的 board_init 函数里面调用 rel_test 函数，否则 rel_reset 不会被编译进 uboot。修改完成后的 mx6ullevk.c 如图 32.2.6.1 所示：

```
826  static int rel_a = 0;
827
828  void rel_test(void)
829  {
830      rel_a = 100;
831      printf("rel_test\r\n");
832  }
833
834  int board_init(void)
835  {
836      rel_test();
```

图 32.2.6.1 加入 rel 测试相关代码

board_init 函数会调用 rel_test，rel_test 会调用全局变量 rel_a，使用如下命令编译 uboot：

```
./mx6ull_alientek_emmc.sh
```

编译完成以后，使用 arm-linux-gnueabihf-objdump 将 u-boot 进行反汇编，得到 u-boot.dis 这个汇编文件，命令如下：

```
arm-linux-gnueabihf-objdump -D -m arm u-boot > u-boot.dis
```

在 u-boot.dis 文件中找到 rel_a、rel_rest 和 board_init，相关内容如下所示：

示例代码 32.2.6.3 汇编文件代码段

```
1  87804184 <rel_test>:
2  87804184:    e59f300c    ldr r3, [pc, #12]   ; 87804198 <rel_test+0x14>
3  87804188:    e3a02064    mov r2, #100        ; 0x64
4  8780418c:    e59f0008    ldr r0, [pc, #8]    ; 8780419c <rel_test+0x18>
5  87804190:    e5832000    str r2, [r3]
6  87804194:    ea00d668    b   87839b3c <printf>
7  87804198:    8785da50            ; <UNDEFINED> instruction: 0x8785da50
8  8780419c:    878426a2    strhi   r2, [r4, r2, lsr #13]
9
10 878041a0 <board_init>:
11 878041a0:    e92d4010    push    {r4, lr}
12 878041a4:    ebfffff6    bl  87804184 <rel_test>
13
14 ......
15
16 8785da50 <rel_a>:
17 8785da50:    00000000    andeq   r0, r0, r0
```

第 12 行是 borad_init 调用 rel_test 函数，用到了 bl 指令，而 bl 指令时位置无关指令，bl 指令是相对寻址的(pc+offset)，因此 uboot 中函数调用是与绝对位置无关的。

再来看一下函数 rel_test 对于全局变量 rel_a 的调用，第 2 行设置 r3 的值为 pc+12 地址处的值，因为 ARM 流水线的原因，pc 寄存器的值为当前地址+8，因此 pc=0X87804184+8=0X8780418C，r3=0X8780418C+12=0X87804198，第 7 行就是 0X87804198 这个地址，0X87804198 处的值为 0X8785DA50。根据第 17 行可知，0X8785DA50 正是变量 rel_a 的地址，最终 r3=0X8785DA50。

第 3 行，r2=100。

第 5 行，将 r2 内的值写到 r3 地址处，也就是设置地址 0X8785DA50 的值为 100，这不就是示例代码代码 32.2.6.2 中的第 5 行：rel_a = 100。

总结一下 rel_a=100 的汇编执行过程：

①、在函数 rel_test 末尾处有一个地址为 0X87804198 的内存空间(示例代码 32.2.6.3 第 7 行)，此内存空间保存着变量 rel_a 的地址。

②、函数 rel_test 要想访问变量 rel_a，首先访问末尾的 0X87804198 来获取变量 rel_a 的地址，而访问 0X87804198 是通过偏移来访问的，很明显是个位置无关的操作。

③、通过 0X87804198 获取到变量 rel_a 的地址，对变量 rel_a 进行操作。

④、可以看出，函数 rel_test 对变量 rel_a 的访问没有直接进行，而是使用了一个第三方偏移地址 0X87804198，专业术语叫做 Label。这个第三方偏移地址就是实现重定位后运行不会出错的重要原因！

uboot 重定位后偏移为 0X18747000，那么重定位后函数 rel_test 的首地址就是 0X87804184+0X18747000=0X9FF4B184。保存变量 rel_a 地址的 Label 就是 0X9FF4B184+8+12=0X9FF4B198(既：0X87804198+0X18747000)，变量 rel_a 的地址就为 0X8785DA50+0X18747000=0X9FFA4A50。重定位后函数 rel_test 要想正常访问变量 rel_a 就得设置 0X9FF4B198(重定位后的 Label)地址出的值为 0X9FFA4A50(重定位后的变量 rel_a 地址)。这样就解决了重定位后链接地址和运行地址不一致的问题。

可以看出，uboot 对于重定位后链接地址和运行地址不一致的解决方法就是采用位置无关码，在使用 ld 进行链接的时候使用选项"-pie"生成位置无关的可执行文件。在文件 arch/arm/config.mk 下有如下代码：

示例代码 32.2.6.4 config.mk 文件代码段

```
82 # needed for relocation
83 LDFLAGS_u-boot += -pie
```

第 83 行就是设置 uboot 链接选项，加入了"-pie"选项，编译链接 uboot 的时候就会使用到"-pie"，如图 32.2.6.2 所示：

```
  arm-linux-gnueabihf-ld.bfd   -pie  --gc-sections -Bstatic -Ttext 0x87800000 -o u-boot -T u-boot.lds arc
h/arm/cpu/armv7/start.o --start-group  arch/arm/cpu/built-in.o  arch/arm/cpu/armv7/built-in.o  arch/arm/i
mx-common/built-in.o  arch/arm/lib/built-in.o  board/freescale/common/built-in.o  board/freescale/mx6ull_
alientek_emmc/built-in.o  cmd/built-in.o  common/built-in.o  disk/built-in.o  drivers/built-in.o  drivers
/dma/built-in.o  drivers/gpio/built-in.o  drivers/i2c/built-in.o  drivers/mmc/built-in.o  drivers/mtd/bui
lt-in.o  drivers/mtd/onenand/built-in.o  drivers/mtd/spi/built-in.o  drivers/net/built-in.o  drivers/net/
phy/built-in.o  drivers/pci/built-in.o  drivers/power/built-in.o  drivers/power/battery/built-in.o  drive
rs/power/fuel_gauge/built-in.o  drivers/power/mfd/built-in.o  drivers/power/pmic/built-in.o  drivers/powe
r/regulator/built-in.o  drivers/serial/built-in.o  drivers/spi/built-in.o  drivers/usb/dwc3/built-in.o  d
rivers/usb/emul/built-in.o  drivers/usb/eth/built-in.o  drivers/usb/gadget/built-in.o  drivers/usb/gadget
/udc/built-in.o  drivers/usb/host/built-in.o  drivers/usb/musb-new/built-in.o  drivers/usb/musb/built-in.
o  drivers/usb/phy/built-in.o  drivers/usb/ulpi/built-in.o  fs/built-in.o  lib/built-in.o  net/built-in.o
  test/built-in.o  test/dm/built-in.o --end-group arch/arm/lib/eabi_compat.o  -L /usr/local/arm/gcc-linar
o-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/bin/../lib/gcc/arm-linux-gnueabihf/4.9.4 -lgcc -Map u-boot.map
  arm-linux-gnueabihf-objcopy --gap-fill=0xff  -j .text -j .secure_text -j .rodata -j .hash -j .data -j .
got -j .got.plt -j .u_boot_list -j .rel.dyn -O binary  u-boot u-boot-nodtb.bin
```

图 32.2.6.2 链接命令

使用"-pie"选项以后会生成一个.rel.dyn 段，uboot 就是靠这个.rel.dyn 来解决重定位问题的，在 u-bot.dis 的.rel.dyn 段中有如下所示内容：

示例代码 32.2.6.5 .rel.dyn 段代码段

```
1 Disassembly of section .rel.dyn:
2
3 8785da44 <__rel_dyn_end-0x8ba0>:
4 8785da44: 87800020    strhi   r0, [r0, r0, lsr #32]
```

```
5 8785da48: 00000017    andeq   r0, r0, r7, lsl r0
6 ......
7 8785dfb4: 87804198           ; <UNDEFINED> instruction: 0x87804198
8 8785dfb8: 00000017    andeq   r0, r0, r7, lsl r0
```

先来看一下.rel.dyn 段的格式，类似第 7 行和第 8 行这样的是一组，也就是两个 4 字节数据为一组。高 4 字节是 Label 地址标识 0X17，低 4 字节就是 Label 的地址，首先判断 Label 地址标识是否正确，也就是判断高 4 字节是否为 0X17，如果是的话高 4 字节就是 Label 值。

第 7 行值为 0X87804198，第 8 行为 0X00000017，说明第 7 行的 0X87804198 是个 Label，这个正是示例代码 32.2.6.3 中存放变量 rel_a 地址的那个 Label。根据前面的分析，只要将地址 0X87804198+offset 处的值改为重定位后的变量 rel_a 地址即可。我们猜测的是否正确，看一下 uboot 对.rel.dyn 段的重定位即可(示例代码代码 32.2.6.1 中的第 94~109 行)，.rel.dyn 段的重定位代码如下：

示例代码 32.2.6.6 relocate.S 代码段

```
91      /*
92       * fix .rel.dyn relocations
93       */
94      ldr r2, =__rel_dyn_start /* r2 <- SRC &__rel_dyn_start */
95      ldr r3, =__rel_dyn_end   /* r3 <- SRC &__rel_dyn_end */
96  fixloop:
97      ldmia   r2!, {r0-r1}     /* (r0,r1) <- (SRC location,fixup) */
98      and r1, r1, #0xff
99      cmp r1, #23              /* relative fixup? */
100     bne fixnext
101
102     /* relative fix: increase location by offset */
103     add r0, r0, r4
104     ldr r1, [r0]
105     add r1, r1, r4
106     str r1, [r0]
107 fixnext:
108     cmp r2, r3
109     blo fixloop
```

第 94 行，r2=__rel_dyn_start，也就是.rel.dyn 段的起始地址。

第 95 行，r3=__rel_dyn_end，也就是.rel.dyn 段的终止地址。

第 97 行，从.rel.dyn 段起始地址开始，每次读取两个 4 字节的数据存放到 r0 和 r1 寄存器中，r0 存放低 4 字节的数据，也就是 Label 地址；r1 存放高 4 字节的数据，也就是 Label 标志。

第 98 行，r1 中给的值与 0xff 进行与运算，其实就是取 r1 的低 8 位。

第 99 行，判断 r1 中的值是否等于 23(0X17)。

第 100 行，如果 r1 不等于 23 的话就说明不是描述 Label 的，执行函数 fixnext，否则的话继续执行下面的代码。

第 103 行，r0 保存着 Label 值，r4 保存着重定位后的地址偏移，r0+r4 就得到了重定位后的 Label 值。此时 r0 保存着重定位后的 Label 值，相当于 0X87804198+0X18747000=0X9FF4B198。

第 104，读取重定位后 Label 所保存的变量地址，此时这个变量地址还是重定位前的(相当

于 rel_a 重定位前的地址 0X8785DA50)，将得到的值放到 r1 寄存器中。

第 105 行，r1+r4 即可得到重定位后的变量地址，相当于 rel_a 重定位后的 0X8785DA50+0X18747000=0X9FFA4A50。

第 106 行，重定位后的变量地址写入到重定位后的 Label 中，相等于设置地址 0X9FF4B198 处的值为 0X9FFA4A50。

第 108 行，比较 r2 和 r3，查看.rel.dyn 段重定位是否完成。

第 109 行，如果 r2 和 r3 不相等，说明.rel.dyn 重定位还未完成，因此跳到 fixloop 继续重定位.rel.dyn 段。

可以看出，uboot 中对.rel.dyn 段的重定位方法和我们猜想的一致。.rel.dyn 段的重定位比较复杂一点，有点绕，因为涉及到链接地址和运行地址的问题。

32.2.7 relocate_vectors 函数详解

函数 relocate_vectors 用于重定位向量表，此函数定义在文件函数源码如下：

示例代码 32.2.7.1 relocate.S 代码段

```
27 ENTRY(relocate_vectors)
28
29 #ifdef CONFIG_CPU_V7M
30     /*
31     * On ARMv7-M we only have to write the new vector address
32     * to VTOR register.
33     */
34     ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr    */
35     ldr r1, =V7M_SCB_BASE
36     str r0, [r1, V7M_SCB_VTOR]
37 #else
38 #ifdef CONFIG_HAS_VBAR
39     /*
40     * If the ARM processor has the security extensions,
41     * use VBAR to relocate the exception vectors.
42     */
43     ldr r0, [r9, #GD_RELOCADDR]     /* r0 = gd->relocaddr    */
44     mcr     p15, 0, r0, c12, c0, 0  /* Set VBAR              */
45 #else
46     /*
47     * Copy the relocated exception vectors to the
48     * correct address
49     * CP15 c1 V bit gives us the location of the vectors:
50     * 0x00000000 or 0xFFFF0000.
51     */
52     ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr            */
53     mrc p15, 0, r2, c1, c0, 0   /* V bit (bit[13]) in CP15 c1   */
54     ands    r2, r2, #(1 << 13)
55     ldreq   r1, =0x00000000       /* If V=0                       */
```

```
56      ldrne   r1, =0xFFFF0000      /* If V=1                      */
57      ldmia   r0!, {r2-r8,r10}
58      stmia   r1!, {r2-r8,r10}
59      ldmia   r0!, {r2-r8,r10}
60      stmia   r1!, {r2-r8,r10}
61 #endif
62 #endif
63      bx  lr
64
65 ENDPROC(relocate_vectors)
```

第 29 行，如果定义了 CONFIG_CPU_V7M 的话就执行第 30~36 行的代码，这是 Cortex-M 内核单片机执行的语句，因此对于 I.MX6ULL 来说是无效的。

第 38 行，如果定义了 CONFIG_HAS_VBAR 的话就执行此语句，这个是向量表偏移，Cortex-A7 是支持向量表偏移的。而且，在.config 里面定义了 CONFIG_HAS_VBAR，因此会执行这个分支。

第 43 行，r0=gd->relocaddr，也就是重定位后 uboot 的首地址，向量表肯定是从这个地址开始存放的。

第 44 行，将 r0 的值写入到 CP15 的 VBAR 寄存器中，也就是将新的向量表首地址写入到寄存器 VBAR 中，设置向量表偏移。

32.2.8 board_init_r 函数详解

第 32.2.5 小节讲解了 board_init_f 函数，在此函数里面会调用一系列的函数来初始化一些外设和 gd 的成员变量。但是 board_init_f 并没有初始化所有的外设，还需要做一些后续工作，这些后续工作就是由函数 board_init_r 来完成的，board_init_r 函数定义在文件 common/board_r.c 中，代码如下：

示例代码 32.2.8.1 board_r.c 代码段

```
991  void board_init_r(gd_t *new_gd, ulong dest_addr)
992  {
993  #ifdef CONFIG_NEEDS_MANUAL_RELOC
994     int i;
995  #endif
996
997  #ifdef CONFIG_AVR32
998     mmu_init_r(dest_addr);
999  #endif
1000
1001 #if !defined(CONFIG_X86) && !defined(CONFIG_ARM)
&& !defined(CONFIG_ARM64)
1002    gd = new_gd;
1003 #endif
1004
1005 #ifdef CONFIG_NEEDS_MANUAL_RELOC
1006    for (i = 0; i < ARRAY_SIZE(init_sequence_r); i++)
```

```
1007        init_sequence_r[i] += gd->reloc_off;
1008 #endif
1009
1010    if (initcall_run_list(init_sequence_r))
1011        hang();
1012
1013    /* NOTREACHED - run_main_loop() does not return */
1014    hang();
1015 }
```

第 1010 行调用 initcall_run_list 函数来执行初始化序列 init_sequence_r，init_sequence_r 是一个函数集合，init_sequence_r 也定义在文件 common/board_r.c 中，由于 init_sequence_f 的内容比较长，里面有大量的条件编译代码，这里为了缩小篇幅，将条件编译部分删除掉了，去掉条件编译以后的 init_sequence_r 定义如下：

示例代码 32.2.8.2 board_r.c 代码段

```
1  init_fnc_t init_sequence_r[] = {
2      initr_trace,
3      initr_reloc,
4      initr_caches,
5      initr_reloc_global_data,
6      initr_barrier,
7      initr_malloc,
8      initr_console_record,
9      bootstage_relocate,
10     initr_bootstage,
11     board_init, /* Setup chipselects */
12     stdio_init_tables,
13     initr_serial,
14     initr_announce,
15     INIT_FUNC_WATCHDOG_RESET
16     INIT_FUNC_WATCHDOG_RESET
17     INIT_FUNC_WATCHDOG_RESET
18     power_init_board,
19     initr_flash,
20     INIT_FUNC_WATCHDOG_RESET
21     initr_nand,
22     initr_mmc,
23     initr_env,
24     INIT_FUNC_WATCHDOG_RESET
25     initr_secondary_cpu,
26     INIT_FUNC_WATCHDOG_RESET
27     stdio_add_devices,
28     initr_jumptable,
29     console_init_r,     /* fully init console as a device */
```

```
30      INIT_FUNC_WATCHDOG_RESET
31      interrupt_init,
32      initr_enable_interrupts,
33      initr_ethaddr,
34      board_late_init,
35      INIT_FUNC_WATCHDOG_RESET
36      INIT_FUNC_WATCHDOG_RESET
37      INIT_FUNC_WATCHDOG_RESET
38      initr_net,
39      INIT_FUNC_WATCHDOG_RESET
40      run_main_loop,
41 };
```

第 2 行，initr_trace 函数，如果定义了宏 CONFIG_TRACE 的话就会调用函数 trace_init，初始化和调试跟踪有关的内容。

第 3 行，initr_reloc 函数用于设置 gd->flags，标记重定位完成。

第 4 行，initr_caches 函数用于初始化 cache，使能 cache。

第 5 行，initr_reloc_global_data 函数，初始化重定位后 gd 的一些成员变量。

第 6 行，initr_barrier 函数，I.MX6ULL 未用到。

第 7 行，initr_malloc 函数，初始化 malloc。

第 8 行，initr_console_record 函数，初始化控制台相关的内容，I.MX6ULL 未用到，空函数。

第 9 行，bootstage_relocate 函数，启动状态重定位。

第 10 行，initr_bootstage 函数，初始化 bootstage 什么的。

第 11 行，board_init 函数，板级初始化，包括 74XX 芯片，I2C、FEC、USB 和 QSPI 等。这里执行的是 mx6ull_alientek_emmc.c 文件中的 board_init 函数。

第 12 行，stdio_init_tables 函数，stdio 相关初始化。

第 13 行，initr_serial 函数，初始化串口。

第 14 行，initr_announce 函数，与调试有关，通知已经在 RAM 中运行。

第 18 行，power_init_board 函数，初始化电源芯片，正点原子的 I.MX6ULL 开发板没有用到。

第 19 行，initr_flash 函数，对于 I.MX6ULL 而言，没有定义宏 CONFIG_SYS_NO_FLASH 的话函数 initr_flash 才有效。但是 mx6_common.h 中定义了宏 CONFIG_SYS_NO_FLASH，所以此函数无效。

第 21 行，initr_nand 函数，初始化 NAND，如果使用 NAND 版本核心板的话就会初始化 NAND。

第 22 行，initr_mmc 函数，初始化 EMMC，如果使用 EMMC 版本核心板的话就会初始化 EMMC，串口输出如图 32.2.8.1 所示信息：

```
512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
```

图 32.2.8.1 EMMC 信息输出

从图 32.2.8.1 可以看出，此时有两个 EMCM 设备，FSL_SDHC:0 和 FSL_SDHC:1。

第 23 行，initr_env 函数，初始化环境变量。

第 25 行，initr_secondary_cpu 函数，初始化其他 CPU 核，I.MX6ULL 只有一个核，因此此函数没用。

第 27 行，stdio_add_devices 函数，各种输入输出设备的初始化，如 LCD driver，I.MX6ULL 使用 drv_video_init 函数初始化 LCD。会输出如图 32.2.8.2 所示信息：

```
Display: ATK-LCD-7-1024x600 (1024x600)
Video: 1024x600x24
```

图 32.2.8.2 LCD 信息

第 28 行，initr_jumptable 函数，初始化跳转表。

第 29 行，console_init_r 函数，控制台初始化，初始化完成以后此函数会调用 stdio_print_current_devices 函数来打印出当前的控制台设备，如图 32.2.8.3 所示：

```
In:    serial
Out:   serial
Err:   serial
```

图 32.2.8.3 控制台信息

第 31 行，interrupt_init 函数，初始化中断。

第 32 行，initr_enable_interrupts 函数，使能中断。

第 33 行，initr_ethaddr 函数，初始化网络地址，也就是获取 MAC 地址。读取环境变量“ethaddr”的值。

第 34 行，board_late_init 函数，板子后续初始化，此函数定义在文件 mx6ull_alientek_emmc.c 中，如果环境变量存储在 EMMC 或者 SD 卡中的话此函数会调用 board_late_mmc_env_init 函数初始化 EMMC/SD。会切换到正在时候用的 emmc 设备，代码如图 32.2.8.4 所示：

```
30 void board_late_mmc_env_init(void)
31 {
32     char cmd[32];
33     char mmcblk[32];
34     u32 dev_no = mmc_get_env_dev();
35 
36     if (!check_mmc_autodetect())
37         return;
38 
39     setenv_ulong("mmcdev", dev_no);
40 
41     /* Set mmcblk env */
42     sprintf(mmcblk, "/dev/mmcblk%dp2 rootwait rw",
43         mmc_map_to_kernel_blk(dev_no));
44     setenv("mmcroot", mmcblk);
45 
46     sprintf(cmd, "mmc dev %d", dev_no);
47     run_command(cmd, 0);
48 }
```

图 32.2.8.4 board_late_mmc_env_init 函数

图 32.2.8.4 中的第 46 行和第 47 行就是运行“mmc dev xx”命令，用于切换到正在使用的 EMMC 设备，串口输出信息如图 32.2.8.5 所示：

```
switch to partitions #0, OK
mmc1(part 0) is current device
```

图 32.2.8.5 切换 mmc 设备

第 38 行，initr_net 函数，初始化网络设备，函数调用顺序为：

initr_net->eth_initialize->board_eth_init()，串口输出如图 32.2.8.6 所示信息：

```
Net:   FEC1
```

图 32.2.8.6 网络信息输出

第 40 行，run_main_loop 行，主循环，处理命令。

32.2.9 run_main_loop 函数详解

uboot 启动以后会进入 3 秒倒计时，如果在 3 秒倒计时结束之前按下按下回车键，那么就会进入 uboot 的命令模式，如果倒计时结束以后都没有按下回车键，那么就会自动启动 Linux 内核，这个功能就是由 run_main_loop 函数来完成的。run_main_loop 函数定义在文件 common/board_r.c 中，函数内容如下：

示例代码 32.2.9.1 board_r.c 文件代码段

```
753 static int run_main_loop(void)
754 {
755     #ifdef CONFIG_SANDBOX
756     sandbox_main_loop_init();
757     #endif
758 /* main_loop() can return to retry autoboot, if so just run it 
again */
759     for (;;)
760             main_loop();
761     return 0;
762 }
```

第 7 行和第 8 行是个死循环，“for(;;)”和“while(1)”功能一样，死循环里面就一个 main_loop 函数，main_loop 函数定义在文件 common/main.c 里面，代码如下：

示例代码 32.2.9.2 main.c 文件代码段

```
43 /* We come here after U-Boot is initialised and ready to process 
commands */
44 void main_loop(void)
45 {
46     const char *s;
47 
48     bootstage_mark_name(BOOTSTAGE_ID_MAIN_LOOP, "main_loop");
49 
50 #ifndef CONFIG_SYS_GENERIC_BOARD
51     puts("Warning: Your board does not use generic board. Please 
read\n");
52     puts("doc/README.generic-board and take action. Boards not\n");
53     puts("upgraded by the late 2014 may break or be removed.\n");
54 #endif
55 
56 #ifdef CONFIG_VERSION_VARIABLE
57     setenv("ver", version_string);  /* set version variable */
58 #endif /* CONFIG_VERSION_VARIABLE */
```

```
59
60     cli_init();
61
62     run_preboot_environment_command();
63
64 #if defined(CONFIG_UPDATE_TFTP)
65     update_tftp(0UL, NULL, NULL);
66 #endif /* CONFIG_UPDATE_TFTP */
67
68     s = bootdelay_process();
69     if (cli_process_fdt(&s))
70         cli_secure_boot_cmd(s);
71
72     autoboot_command(s);
73
74     cli_loop();
75 }
```

第 48 行，调用 bootstage_mark_name 函数，打印出启动进度。

第 57 行，如果定义了宏 CONFIG_VERSION_VARIABLE 的话就会执行函数 setenv，设置换将变量 ver 的值为 version_string，也就是设置版本号环境变量。version_string 定义在文件 cmd/version.c 中，定义如下：

```
const char __weak version_string[] = U_BOOT_VERSION_STRING;
```

U_BOOT_VERSION_STRING 是个宏，定义在文件 include/version.h，如下：

```
#define U_BOOT_VERSION_STRING U_BOOT_VERSION " (" U_BOOT_DATE " - " \
    U_BOOT_TIME " " U_BOOT_TZ ")" CONFIG_IDENT_STRING
```

U_BOOT_VERSION 定义在文件 include/generated/version_autogenerated.h 中，文件 version_autogenerated.h 内如如下：

示例代码 32.2.9.4 version_autogenerated.h 文件代码

```
1 #define PLAIN_VERSION "2016.03"
2 #define U_BOOT_VERSION "U-Boot " PLAIN_VERSION
3 #define CC_VERSION_STRING "arm-linux-gnueabihf-gcc (Linaro GCC 4.9-2017.01) 4.9.4"
4 #define LD_VERSION_STRING "GNU ld (Linaro_Binutils-2017.01) 2.24.0.20141017 Linaro 2014_11-3-git"
```

可以看出，U_BOOT_VERSION 为“U-boot 2016.03”，U_BOOT_DATE 、 U_BOOT_TIME 和 U_BOOT_TZ 这定义在文件 include/generated/timestamp_autogenerated.h 中，如下所示：

示例代码 32.2.9.5 timestamp_autogenerated.h 文件代码

```
1 #define U_BOOT_DATE "Apr 25 2019"
2 #define U_BOOT_TIME "21:10:53"
3 #define U_BOOT_TZ "+0800"
4 #define U_BOOT_DMI_DATE "04/25/2019"
```

宏 CONFIG_IDENT_STRING 为空，所以 U_BOOT_VERSION_STRING 为"U-Boot 2016.03 (Apr 25 2019 - 21:10:53 +0800)"，进入 uboot 命令模式，输入命令"version"查看版本号，如图 32.2.9.1 所示：

```
U-Boot 2016.03 (Apr 25 2019 - 21:10:53 +0800)
arm-linux-gnueabihf-gcc (Linaro GCC 4.9-2017.01) 4.9.4
GNU ld (Linaro_Binutils-2017.01) 2.24.0.20141017 Linaro 2014_11-3-git
=>
```

图 32.2.9.1 版本查询

图 32.2.9.1 中的第一行就是 uboot 版本号，和我们分析的一致。

接着回到示例代码 32.2.9.2 中，第 60 行，cli_init 函数，跟命令初始化有关，初始化 hush shell 相关的变量。

第 62 行，run_preboot_environment_command 函数，获取环境变量 perboot 的内容，perboot 是一些预启动命令，一般不使用这个环境变量。

第 68 行，bootdelay_process 函数，此函数会读取环境变量 bootdelay 和 bootcmd 的内容，然后将 bootdelay 的值赋值给全局变量 stored_bootdelay，返回值为环境变量 bootcmd 的值。

第 69 行，如果定义了 CONFIG_OF_CONTROL 的话函数 cli_process_fdt 就会实现，如果没有定义 CONFIG_OF_CONTROL 的话函数 cli_process_fdt 直接返回一个 false。在本 uboot 中没有定义 CONFIG_OF_CONTROL，因此 cli_process_fdt 函数返回值为 false。

第 72 行，autoboot_command 函数，此函数就是检查倒计时是否结束？倒计时结束之前有没有被打断？此函数定义在文件 common/autoboot.c 中，内容如下：

示例代码 32.2.9.5 auboboot.c 文件代码段

```
380 void autoboot_command(const char *s)
381 {
382     debug("### main_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");
383 
384     if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay))
{
385 #if defined(CONFIG_AUTOBOOT_KEYED)
&& !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
386         int prev = disable_ctrlc(1);    /* disable Control C checking
*/
387 #endif
388 
389         run_command_list(s, -1, 0);
390 
391 #if defined(CONFIG_AUTOBOOT_KEYED)
&& !defined(CONFIG_AUTOBOOT_KEYED_CTRLC)
392         disable_ctrlc(prev);    /* restore Control C checking */
393 #endif
394     }
395 
396 #ifdef CONFIG_MENUKEY
397     if (menukey == CONFIG_MENUKEY) {
398         s = getenv("menucmd");
```

```
399         if (s)
400             run_command_list(s, -1, 0);
401     }
402 #endif /* CONFIG_MENUKEY */
403 }
```

可以看出，autoboot_command 函数里面有很多条件编译，条件编译一多就不利于我们阅读程序（所以正点原子的例程基本是不用条件编译的，就是为了方便大家阅读源码）！宏 CONFIG_AUTOBOOT_KEYED 、 CONFIG_AUTOBOOT_KEYED_CTRLC 和 CONFIG_MENUKEY 这三个宏在 I.MX6ULL 里面没有定义，所以讲示例代码 32.2.9.5 进行精简，得到如下代码：

示例代码 32.2.9.6 autoboot_command 函数精简版本

```
1 void autoboot_command(const char *s)
2 {
3   if (stored_bootdelay != -1 && s && !abortboot(stored_bootdelay)) {
4       run_command_list(s, -1, 0);
5   }
6 }
```

当一下三条全部成立的话，就会执行函数 run_command_list。

①、stored_bootdelay 不等于-1。

②、s 不为空。

③、函数 abortboot 返回值为 0。

stored_bootdelay 等于环境变量 bootdelay 的值；s 是环境变量 bootcmd 的值，一般不为空，因此前两个成立，就剩下了函数 abortboot 的返回值，abortboot 函数也定义在文件 common/autoboot.c 中，内容如下：

示例代码 32.2.9.7 abortboot 函数

```
283 static int abortboot(int bootdelay)
284 {
285 #ifdef CONFIG_AUTOBOOT_KEYED
286     return abortboot_keyed(bootdelay);
287 #else
288     return abortboot_normal(bootdelay);
289 #endif
290 }
```

因为宏 CONFIG_AUTOBOOT_KEYE 未定义，因此执行函数 abortboot_normal，好吧，绕来绕去的！接着来看函数 abortboot_normal，此函数也定义在文件 common/autoboot.c 中，内容如下：

示例代码 32.2.9.8 abortboot_normal 函数

```
225 static int abortboot_normal(int bootdelay)
226 {
227     int abort = 0;
228     unsigned long ts;
229 
230 #ifdef CONFIG_MENUPROMPT
```

```
231     printf(CONFIG_MENUPROMPT);
232 #else
233     if (bootdelay >= 0)
234         printf("Hit any key to stop autoboot: %2d ", bootdelay);
235 #endif
236
237 #if defined CONFIG_ZERO_BOOTDELAY_CHECK
238     /*
239      * Check if key already pressed
240      * Don't check if bootdelay < 0
241      */
242     if (bootdelay >= 0) {
243         if (tstc()) {   /* we got a key press   */
244             (void) getc();  /* consume input    */
245             puts("\b\b\b 0");
246             abort = 1;  /* don't auto boot  */
247         }
248     }
249 #endif
250
251     while ((bootdelay > 0) && (!abort)) {
252         --bootdelay;
253         /* delay 1000 ms */
254         ts = get_timer(0);
255         do {
256             if (tstc()) {   /* we got a key press   */
257                 abort  = 1; /* don't auto boot  */
258                 bootdelay = 0;  /* no more delay    */
259 # ifdef CONFIG_MENUKEY
260                 menukey = getc();
261 # else
262                 (void) getc();  /* consume input    */
263 # endif
264                 break;
265             }
266             udelay(10000);
267         } while (!abort && get_timer(ts) < 1000);
268
269         printf("\b\b\b%2d ", bootdelay);
270     }
271
272     putc('\n');
273
```

```
274 #ifdef CONFIG_SILENT_CONSOLE
275     if (abort)
276         gd->flags &= ~GD_FLG_SILENT;
277 #endif
278
279     return abort;
280 }
```

函数 abortboot_normal 同样很多条件编译，删除掉条件编译相关代码后 abortboot_normal 函数内容如下：

示例代码 32.2.9.9 abortboot_normal 函数精简

```
1  static int abortboot_normal(int bootdelay)
2  {
3      int abort = 0;
4      unsigned long ts;
5
6      if (bootdelay >= 0)
7          printf("Hit any key to stop autoboot: %2d ", bootdelay);
8
9      while ((bootdelay > 0) && (!abort)) {
10         --bootdelay;
11         /* delay 1000 ms */
12         ts = get_timer(0);
13         do {
14             if (tstc()) {   /* we got a key press   */
15                 abort  = 1; /* don't auto boot  */
16                 bootdelay = 0;  /* no more delay    */
17                 (void) getc();  /* consume input    */
18                 break;
19             }
20             udelay(10000);
21         } while (!abort && get_timer(ts) < 1000);
22
23         printf("\b\b\b%2d ", bootdelay);
24     }
25     putc('\n');
26     return abort;
27 }
```

第 3 行的变量 abort 是函数 abortboot_normal 的返回值，默认值为 0。

第 7 行通过串口输出“Hit any key to stop autoboot”字样，如图 32.2.9.2 所示：

```
Hit any key to stop autoboot:  0
```

图 32.2.9.2 倒计时

第 9~21 行就是倒计时的具体实现。

第 14 行判断键盘是否有按下，也就是是否打断了倒计时，如果键盘按下的话就执行相应的分支。比如设置 abort 为 1，设置 bootdelay 为 0 等，最后跳出倒计时循环。

第 26 行，返回 abort 的值，如果倒计时自然结束，没有被打断 abort 就为 0，否则的话 abort 的值就为 1。

回到示例代码 32.2.9.6 的 autoboot_command 函数中，如果倒计时自然结束那么就执行函数 run_command_list，此函数会执行参数 s 指定的一系列命令，也就是环境变量 bootcmd 的命令，bootcmd 里面保存着默认的启动命令，因此 linux 内核启动！这个就是 uboot 中倒计时结束以后自动启动 linux 内核的原理。如果倒计时结束之前按下了键盘上的按键，那么 run_command_list 函数就不会执行，相当于 autoboot_command 是个空函数。

回到"遥远"的示例代码 32.2.9.2 中的 main_loop 函数中，如果倒计时结束之前按下按键，那么就会执行第 74 行的 cli_loop 函数，这个就是命令处理函数，负责接收好处理输入的命令。

32.2.10 cli_loop 函数详解

cli_loop 函数是 uboot 的命令行处理函数，我们在 uboot 中输入各种命令，进行各种操作就是有 cli_loop 来处理的，此函数定义在文件 common/cli.c 中，函数内容如下：

示例代码 32.2.10.1 cli.c 文件代码段

```
202 void cli_loop(void)
203 {
204 #ifdef CONFIG_SYS_HUSH_PARSER
205     parse_file_outer();
206     /* This point is never reached */
207     for (;;);
208 #else
209     cli_simple_loop();
210 #endif /*CONFIG_SYS_HUSH_PARSER*/
211 }
```

在文件 include/configs/mx6_common.h 中有定义宏 CONFIG_SYS_HUSH_PARSER，而正点原子的 I.MX6ULL 开发板配置头文件 mx6ullevk.h 里面会引用 mx_common.h 这个头文件，因此宏 CONFIG_SYS_HUSH_PARSER 有定义。

第 205 行调用函数 parse_file_outer。

第 207 行是个死循环，永远不会执行到这里。

函数 parse_file_outer 定义在文件 common/cli_hush.c 中，去掉条件编译内容以后的函数内容如下：

示例代码 32.2.10.2 parse_file_outer 函数精简

```
1 int parse_file_outer(void)
2 {
3       int rcode;
4       struct in_str input;
5 
6       setup_file_in_str(&input);
7       rcode = parse_stream_outer(&input, FLAG_PARSE_SEMICOLON);
8       return rcode;
9 }
```

第 3 行调用函数 setup_file_in_str 初始化变量 input 的成员变量。

第 4 行调用函数 parse_stream_outer，这个函数就是 hush shell 的命令解释器，负责接收命令行输入，然后解析并执行相应的命令，函数 parse_stream_outer 定义在文件 common/cli_hush.c 中，精简版的函数内容如下：

示例代码 32.2.10.3 parse_stream_outer 函数精简

```
1  static int parse_stream_outer(struct in_str *inp, int flag)
2  {
3   struct p_context ctx;
4   o_string temp=NULL_O_STRING;
5   int rcode;
6   int code = 1;
7   do {
8       ......
9       rcode = parse_stream(&temp, &ctx, inp,
10                   flag & FLAG_CONT_ON_NEWLINE ? -1 : '\n');
11      ......
12      if (rcode != 1 && ctx.old_flag == 0) {
13          ......
14          run_list(ctx.list_head);
15          ......
16      } else {
17          ......
18      }
19      b_free(&temp);
20  /* loop on syntax errors, return on EOF */
21  } while (rcode != -1 && !(flag & FLAG_EXIT_FROM_LOOP) &&
22      (inp->peek != static_peek || b_peek(inp)));
23  return 0;
24 }
```

第 7~21 行中的 do-while 循环就是处理输入命令的。

第 9 行调用函数 parse_stream 进行命令解析。

第 14 行调用调用 run_list 函数来执行解析出来的命令。

函数 run_list 会经过一系列的函数调用，最终通过调用 cmd_process 函数来处理命令，过程如下：

示例代码 32.2.10.4 run_list 执行流程

```
1  static int run_list(struct pipe *pi)
2  {
3      int rcode=0;
4
5      rcode = run_list_real(pi);
6      ......
7      return rcode;
8  }
```

```
9
10 static int run_list_real(struct pipe *pi)
11 {
12     char *save_name = NULL;
13     ......
14     int if_code=0, next_if_code=0;
15     ......
16     rcode = run_pipe_real(pi);
17     ......
18     return rcode;
19 }
20
21 static int run_pipe_real(struct pipe *pi)
22 {
23     int i;
24
25     int nextin;
26     int flag = do_repeat ? CMD_FLAG_REPEAT : 0;
27     struct child_prog *child;
28     char *p;
29     ......
30     if (pi->num_progs == 1) child = & (pi->progs[0]);
31         ......
32         return rcode;
33     } else if (pi->num_progs == 1 && pi->progs[0].argv != NULL) {
34         ......
35         /* Process the command */
36         return cmd_process(flag, child->argc, child->argv,
37                &flag_repeat, NULL);
38     }
39
40     return -1;
41 }
```

第 5 行，run_list 调用 run_list_real 函数。

第 16 行，run_list_real 函数调用 run_pipe_real 函数。

第 36 行，run_pipe_real 函数调用 cmd_process 函数。

最终通过函数 cmd_process 来处理命令，接下来就是分析 cmd_process 函数。

32.2.11 cmd_process 函数详解

在学习 cmd_process 之前先看一下 uboot 中命令是如何定义的。uboot 使用宏 U_BOOT_CMD 来定义命令，宏 U_BOOT_CMD 定义在文件 include/command.h 中，定义如下：

示例代码 32.2.11.1 U_BOOT_CMD 宏定义

```
#define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help)     \
```

```
    U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, 
NULL)
```

可以看出 U_BOOT_CMD 是 U_BOOT_CMD_COMPLETE 的特例，将 U_BOOT_CMD_COMPLETE 的最后一个参数设置成 NULL 就是 U_BOOT_CMD。宏 U_BOOT_CMD_COMPLETE 如下：

示例代码 32.2.11.2 U_BOOT_CMD_COMPLETE 宏定义

```
#define U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, 
_comp) \
    ll_entry_declare(cmd_tbl_t, _name, cmd) =         \
        U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,  \
                        _usage, _help, _comp);
```

宏 U_BOOT_CMD_COMPLETE 又用到了 ll_entry_declare 和 U_BOOT_CMD_MKENT_COMPLETE。ll_entry_declar 定义在文件 include/linker_lists.h 中，定义如下：

示例代码 32.2.11.3 ll_entry_declare 宏定义

```
#define ll_entry_declare(_type, _name, _list)             \
    _type _u_boot_list_2_##_list##_2_##_name __aligned(4)       \
            __attribute__((unused,                \
            section(".u_boot_list_2_"#_list"_2_"#_name)))
```

_type 为 cmd_tbl_t，因此 ll_entry_declare 就是定义了一个 cmd_tbl_t 变量，这里用到了 C 语言中的“##”连接符符。其中的“##_list”表示用_list 的值来替换，“##_name”就是用_name 的值来替换。

宏 U_BOOT_CMD_MKENT_COMPLETE 定义在文件 include/command.h 中，内容如下：

示例代码 32.2.11.4 U_BOOT_CMD_MKENT_COMPLETE 宏定义

```
#define U_BOOT_CMD_MKENT_COMPLETE(_name, _maxargs, _rep, _cmd,      \
                _usage, _help, _comp)            \
        { #_name, _maxargs, _rep, _cmd, _usage,          \
            _CMD_HELP(_help) _CMD_COMPLETE(_comp) }
```

上述代码中的“#”表示将_name 传递过来的值字符串化，U_BOOT_CMD_MKENT_COMPLETE 又用到了宏_CMD_HELP 和_CMD_COMPLETE，这两个宏的定义如下：

示例代码 32.2.11.5 _CMD_HELP 和_CMD_COMPLETE 宏定义

```
1  #ifdef CONFIG_AUTO_COMPLETE
2  # define _CMD_COMPLETE(x) x,
3  #else
4  # define _CMD_COMPLETE(x)
5  #endif
6  #ifdef CONFIG_SYS_LONGHELP
7  # define _CMD_HELP(x) x,
8  #else
9  # define _CMD_HELP(x)
10 #endif
```

可以看出，如果定义了宏 CONFIG_AUTO_COMPLETE 和 CONFIG_SYS_LONGHELP 的

话，_CMD_COMPLETE 和_CMD_HELP 就是取自身的值，然后在加上一个‘,’。CONFIG_AUTO_COMPLETE 和 CONFIG_SYS_LONGHELP 这两个宏有定义在文件 mx6_common.h 中。

U_BOOT_CMD 宏的流程我们已经清楚了(一个 U_BOOT_CMD 宏就如此的绕来绕去的!)，我们就以一个具体的命令为例，来看一下 U_BOOT_CMD 经过展开以后究竟是个什么模样的。以命令 dhcp 为例，dhcp 命令定义如下：

示例代码 32.2.11.6 dhcp 命令宏定义

```
U_BOOT_CMD(
    dhcp,   3,  1,  do_dhcp,
    "boot image via network using DHCP/TFTP protocol",
    "[loadAddress] [[hostIPaddr:]bootfilename]"
);
```

将其展开，结果如下：

示例代码 32.2.11.7 dhcp 命令展开

```
U_BOOT_CMD(
    dhcp,   3,  1,  do_dhcp,
    "boot image via network using DHCP/TFTP protocol",
    "[loadAddress] [[hostIPaddr:]bootfilename]"
);

1、将 U_BOOT_CMD 展开后为:
U_BOOT_CMD_COMPLETE(dhcp, 3, 1, do_dhcp,
                    "boot image via network using DHCP/TFTP protocol",
                    "[loadAddress] [[hostIPaddr:]bootfilename]",
                    NULL)

2、将 U_BOOT_CMD_COMPLETE 展开后为:
ll_entry_declare(cmd_tbl_t, dhcp, cmd) =          \
U_BOOT_CMD_MKENT_COMPLETE(dhcp, 3, 1, do_dhcp,  \
                 "boot image via network using DHCP/TFTP protocol", \
                 "[loadAddress] [[hostIPaddr:]bootfilename]", \
                 NULL);

3、将 ll_entry_declare 和 U_BOOT_CMD_MKENT_COMPLETE 展开后为:
cmd_tbl_t _u_boot_list_2_cmd_2_dhcp __aligned(4)      \
          __attribute__((unused,section(.u_boot_list_2_cmd_2_dhcp))) \
       { "dhcp", 3, 1, do_dhcp,  \
        "boot image via network using DHCP/TFTP protocol", \
        "[loadAddress] [[hostIPaddr:]bootfilename]",\
        NULL}
```

从示例代码 32.2.11.7 可以看出，dhcp 命令最终展开结果为:

示例代码 32.2.11.8 dhcp 命令最终结果

```
1 cmd_tbl_t _u_boot_list_2_cmd_2_dhcp __aligned(4)      \
```

```
2        __attribute__((unused,section(.u_boot_list_2_cmd_2_dhcp))) \
3        { "dhcp", 3, 1, do_dhcp,  \
4        "boot image via network using DHCP/TFTP protocol", \
5        "[loadAddress] [[hostIPaddr:]bootfilename]",\
6        NULL}
```

第 1 行定义了一个 cmd_tbl_t 类型的变量，变量名为_u_boot_list_2_cmd_2_dhcp，此变量 4 字节对齐。

第 2 行，使用__attribute__关键字设置变量_u_boot_list_2_cmd_2_dhcp 存储在.u_boot_list_2_cmd_2_dhcp 段中。u-boot.lds 链接脚本中有一个名为“.u_boot_list”的段，所有.u_boot_list 开头的段都存放到.u_boot.list 中，如图 32.2.11.1 所示：

```
21  . = ALIGN(4);
22  . = .;
23  . = ALIGN(4);
24  .u_boot_list : {
25   KEEP(*(SORT(.u_boot_list*)));
26  }
```

图 32.2.11.1 u-boot.lds 中的.u_boot_list 段

因此，第 2 行就是设置变量_u_boot_list_2_cmd_2_dhcp 的存储位置。

第 3~6 行，cmd_tbl_t 是个结构体，因此第 3-6 行是初始化 cmd_tbl_t 这个结构体的各个成员变量。cmd_tbl_t 结构体定义在文件 include/command.h 中，内容如下：

示例代码 32.2.11.9 cmd_tbl_t 结构体

```
30 struct cmd_tbl_s {
31     char    *name;          /* Command Name                  */
32     int     maxargs;        /* maximum number of arguments   */
33     int     repeatable;     /* autorepeat allowed?           */
34     /* Implementation function  */
35     int     (*cmd)(struct cmd_tbl_s *, int, int, char * const []);
36     char        *usage;     /* Usage message    (short)      */
37 #ifdef  CONFIG_SYS_LONGHELP
38     char        *help;      /* Help  message    (long)       */
39 #endif
40 #ifdef CONFIG_AUTO_COMPLETE
41                             /* do auto completion on the arguments */
42     int     (*complete)(int argc, char * const argv[], char
last_char, int maxv, char *cmdv[]);
43 #endif
44 };
45
46 typedef struct cmd_tbl_s cmd_tbl_t;
```

结合实例代码 32.2.11.8，可以得出变量_u_boot_list_2_cmd_2_dhcp 的各个成员的值如下所示：

```
_u_boot_list_2_cmd_2_dhcp.name = "dhcp"
_u_boot_list_2_cmd_2_dhcp.maxargs = 3
_u_boot_list_2_cmd_2_dhcp.repeatable = 1
```

```
_u_boot_list_2_cmd_2_dhcp.cmd = do_dhcp
_u_boot_list_2_cmd_2_dhcp.usage = "boot image via network using DHCP/TFTP protocol"
_u_boot_list_2_cmd_2_dhcp.help =   "[loadAddress] [[hostIPaddr:]bootfilename]"
_u_boot_list_2_cmd_2_dhcp.complete = NULL
```

当我们在 uboot 的命令行中输入“dhcp”这个命令的时候，最终执行的是 do_dhcp 这个函数。总结一下，uboot 中使用 U_BOOT_CMD 来定义一个命令，最终的目的就是为了定义一个 cmd_tbl_t 类型的变量，并初始化这个变量的各个成员。uboot 中的每个命令都存储在.u_boot_list 段中，每个命令都有一个名为 do_xxx(xxx 为具体的命令名)的函数，这个 do_xxx 函数就是具体的命令处理函数。

了解了 uboot 中命令的组成以后，再来看一下 cmd_process 函数的处理过程，cmd_process 函数定义在文件 common/command.c 中，函数内容如下：

示例代码 32.2.11.10 command.c 文件代码段

```
500 enum command_ret_t cmd_process(int flag, int argc,
501                 char * const argv[],int *repeatable, ulong *ticks)
502 {
503     enum command_ret_t rc = CMD_RET_SUCCESS;
504     cmd_tbl_t *cmdtp;
505 
506     /* Look up command in command table */
507     cmdtp = find_cmd(argv[0]);
508     if (cmdtp == NULL) {
509         printf("Unknown command '%s' - try 'help'\n", argv[0]);
510         return 1;
511     }
512 
513     /* found - check max args */
514     if (argc > cmdtp->maxargs)
515         rc = CMD_RET_USAGE;
516 
517 #if defined(CONFIG_CMD_BOOTD)
518     /* avoid "bootd" recursion */
519     else if (cmdtp->cmd == do_bootd) {
520         if (flag & CMD_FLAG_BOOTD) {
521             puts("'bootd' recursion detected\n");
522             rc = CMD_RET_FAILURE;
523         } else {
524             flag |= CMD_FLAG_BOOTD;
525         }
526     }
527 #endif
528 
529     /* If OK so far, then do the command */
530     if (!rc) {
```

```
531         if (ticks)
532             *ticks = get_timer(0);
533         rc = cmd_call(cmdtp, flag, argc, argv);
534         if (ticks)
535             *ticks = get_timer(*ticks);
536         *repeatable &= cmdtp->repeatable;
537     }
538     if (rc == CMD_RET_USAGE)
539         rc = cmd_usage(cmdtp);
540     return rc;
541 }
```

第 507 行，调用函数 find_cmd 在命令表中找到指定的命令，find_cmd 函数内容如下：

示例代码 32.2.11.10 command.c 文件代码段

```
118 cmd_tbl_t *find_cmd(const char *cmd)
119 {
120     cmd_tbl_t *start = ll_entry_start(cmd_tbl_t, cmd);
121     const int len = ll_entry_count(cmd_tbl_t, cmd);
122     return find_cmd_tbl(cmd, start, len);
123 }
```

参数 cmd 就是所查找的命令名字，uboot 中的命令表其实就是 cmd_tbl_t 结构体数组，通过函数 ll_entry_start 得到数组的第一个元素，也就是命令表起始地址。通过函数 ll_entry_count 得到数组长度，也就是命令表的长度。最终通过函数 find_cmd_tbl 在命令表中找到所需的命令，每个命令都有一个 name 成员，所以将参数 cmd 与命令表中每个成员的 name 字段都对比一下，如果相等的话就说明找到了这个命令，找到以后就返回这个命令。

回到示例代码 32.2.11.10 的 cmd_process 函数中，找到命令以后肯定就要执行这个命令了，第 533 行调用函数 cmd_call 来执行具体的命令，cmd_call 函数内容如下：

示例代码 32.2.11.11 command.c 文件代码段

```
490 static int cmd_call(cmd_tbl_t *cmdtp, int flag, int argc, char *
const argv[])
491 {
492     int result;
493 
494     result = (cmdtp->cmd)(cmdtp, flag, argc, argv);
495     if (result)
496         debug("Command failed, result=%d\n", result);
497     return result;
498 }
```

在前面的分析中我们知道，cmd_tbl_t 的 cmd 成员就是具体的命令处理函数，所以第 494 行调用 cmdtp 的 cmd 成员来处理具体的命令，返回值为命令的执行结果。

cmd_process 中会检测 cmd_tbl 的返回值，如果返回值为 CMD_RET_USAGE 的话就会调用 cmd_usage 函数输出命令的用法，其实就是输出 cmd_tbl_t 的 usage 成员变量。

32.3 bootz 启动 Linux 内核过程

32.3.1 images 全局变量

不管是 bootz 还是 bootm 命令，在启动 Linux 内核的时候都会用到一个重要的全局变量：images，images 在文件 cmd/bootm.c 中有如下定义：

示例代码 32.3.1.1 images 全局变量

```
43 bootm_headers_t images;    /* pointers to os/initrd/fdt images */
```

images 是 bootm_headers_t 类型的全局变量，bootm_headers_t 是个 boot 头结构体，在文件 include/image.h 中的定义如下(删除了一些条件编译代码)：

示例代码 32.3.1.2 bootm_headers_t 结构体

```
304 typedef struct bootm_headers {
305    /*
306     * Legacy os image header, if it is a multi component image
307     * then boot_get_ramdisk() and get_fdt() will attempt to get
308     * data from second and third component accordingly.
309     */
310    image_header_t  *legacy_hdr_os;     /* image header pointer */
311    image_header_t  legacy_hdr_os_copy; /* header copy */
312    ulong       legacy_hdr_valid;
313
......
333
334 #ifndef USE_HOSTCC
335    image_info_t os;                /* OS 镜像信息             */
336    ulong       ep;                 /* OS 入口点               */
337
338    ulong       rd_start, rd_end;   /* ramdisk 开始和结束位置  */
339
340    char        *ft_addr;           /* 设备树地址              */
341    ulong       ft_len;             /* 设备树长度              */
342
343    ulong       initrd_start;       /* initrd 开始位置         */
344    ulong       initrd_end;         /* initrd 结束位置         */
345    ulong       cmdline_start;      /* cmdline 开始位置        */
346    ulong       cmdline_end;        /* cmdline 结束位置        */
347    bd_t        *kbd;
348 #endif
349
350    int     verify;     /* getenv("verify")[0] != 'n' */
351
352 #define BOOTM_STATE_START       (0x00000001)
353 #define BOOTM_STATE_FINDOS      (0x00000002)
354 #define BOOTM_STATE_FINDOTHER   (0x00000004)
```

```
355 #define BOOTM_STATE_LOADOS      (0x00000008)
356 #define BOOTM_STATE_RAMDISK     (0x00000010)
357 #define BOOTM_STATE_FDT         (0x00000020)
358 #define BOOTM_STATE_OS_CMDLINE  (0x00000040)
359 #define BOOTM_STATE_OS_BD_T     (0x00000080)
360 #define BOOTM_STATE_OS_PREP     (0x00000100)
361 #define BOOTM_STATE_OS_FAKE_GO  (0x00000200)/*'Almost' run the OS*/
362 #define BOOTM_STATE_OS_GO       (0x00000400)
363     int     state;
364 
365 #ifdef CONFIG_LMB
366     struct lmb  lmb;    /* 内存管理相关，不深入研究 */
367 #endif
368 } bootm_headers_t;
```

第 335 行的 os 成员变量是 image_info_t 类型的，为系统镜像信息。

第 352~362 行这 11 个宏定义表示 BOOT 的不同阶段。

接下来看一下结构体 image_info_t，也就是系统镜像信息结构体，此结构体在文件 include/image.h 中的定义如下：

示例代码 32.3.1.3 image_info_t 结构体

```
292 typedef struct image_info {
293     ulong  start, end;              /* blob 开始和结束位置*/
294     ulong  image_start, image_len; /* 镜像起始地址(包括 blob)和长度 */
295     ulong  load;                    /* 系统镜像加载地址*/
296     uint8_t    comp, type, os;      /* 镜像压缩、类型，OS 类型 */
297     uint8_t    arch;                /* CPU 架构 */
298 } image_info_t;
```

全局变量 images 会在 bootz 命令的执行中频繁使用到，相当于 Linux 内核启动的“灵魂”。

32.3.2 do_bootz 函数

bootz 命令的执行函数为 do_bootz，在文件 cmd/bootm.c 中有如下定义：

示例代码 32.3.2.1 do_bootz 函数

```
622 int do_bootz(cmd_tbl_t *cmdtp, int flag, int argc, char * const 
argv[])
623 {
624     int ret;
625 
626     /* Consume 'bootz' */
627     argc--; argv++;
628 
629     if (bootz_start(cmdtp, flag, argc, argv, &images))
630         return 1;
631 
632     /*
```

```
633     * We are doing the BOOTM_STATE_LOADOS state ourselves, so must
634     * disable interrupts ourselves
635     */
636    bootm_disable_interrupts();
637
638    images.os.os = IH_OS_LINUX;
639    ret = do_bootm_states(cmdtp, flag, argc, argv,
640                BOOTM_STATE_OS_PREP | BOOTM_STATE_OS_FAKE_GO |
641                BOOTM_STATE_OS_GO,
642                &images, 1);
643
644    return ret;
645 }
```

第 629 行，调用 bootz_start 函数，bootz_start 函数执行过程参考 32.3.3 小节。

第 636 行，调用函数 bootm_disable_interrupts 关闭中断。

第 638 行，设置 images.os.os 为 IH_OS_LINUX，也就是设置系统镜像为 Linux，表示我们要启动的是 Linux 系统！后面会用到 images.os.os 来挑选具体的启动函数。

第 639 行，调用函数 do_bootm_states 来执行不同的 BOOT 阶段，这里要执行的 BOOT 阶段有:BOOTM_STATE_OS_PREP 、BOOTM_STATE_OS_FAKE_GO 和 BOOTM_STATE_OS_GO。

32.3.3 bootz_start 函数

bootz_srart 函数也定义在文件 cmd/bootm.c 中，函数内容如下：

示例代码 32.3.3.1 bootz_start 函数

```
578 static int bootz_start(cmd_tbl_t *cmdtp, int flag, int argc,
579             char * const argv[], bootm_headers_t *images)
580 {
581    int ret;
582    ulong zi_start, zi_end;
583
584    ret = do_bootm_states(cmdtp, flag, argc, argv,
585                     BOOTM_STATE_START,  images, 1);
586
587    /* Setup Linux kernel zImage entry point */
588    if (!argc) {
589        images->ep = load_addr;
590        debug("*  kernel: default image load address = 0x%08lx\n",
591              load_addr);
592    } else {
593        images->ep = simple_strtoul(argv[0], NULL, 16);
594        debug("*  kernel: cmdline image address = 0x%08lx\n",
595            images->ep);
596    }
597
```

```
598     ret = bootz_setup(images->ep, &zi_start, &zi_end);
599     if (ret != 0)
600         return 1;
601 
602     lmb_reserve(&images->lmb, images->ep, zi_end - zi_start);
603 
604     /*
605      * Handle the BOOTM_STATE_FINDOTHER state ourselves as we do not
606      * have a header that provide this informaiton.
607      */
608     if (bootm_find_images(flag, argc, argv))
609         return 1;
610 
......
619     return 0;
620 }
```

第 584 行，调用函数 do_bootm_states，执行 BOOTM_STATE_START 阶段。

第 593 行，设置 images 的 ep 成员变量，也就是系统镜像的入口点，使用 bootz 命令启动系统的时候就会设置系统在 DRAM 中的存储位置，这个存储位置就是系统镜像的入口点，因此 images->ep=0X80800000。

第 598 行，调用 bootz_setup 函数，此函数会判断当前的系统镜像文件是否为 Linux 的镜像文件，并且会打印出镜像相关信息，bootz_setup 函数稍后会讲解。

第 608 行，调用函数 bootm_find_images 查找 ramdisk 和设备树(dtb)文件，但是我们没有用到 ramdisk，因此此函数在这里仅仅用于查找设备树(dtb)文件，此函数稍后也会讲解。

先来看一下 bootz_setup 函数，此函数定义在文件 arch/arm/lib/bootm.c 中，函数内容如下：

示例代码 32.3.3.2 bootz_setup 函数

```
370 #define LINUX_ARM_ZIMAGE_MAGIC  0x016f2818
371 
372 int bootz_setup(ulong image, ulong *start, ulong *end)
373 {
374     struct zimage_header *zi;
375 
376     zi = (struct zimage_header *)map_sysmem(image, 0);
377     if (zi->zi_magic != LINUX_ARM_ZIMAGE_MAGIC) {
378         puts("Bad Linux ARM zImage magic!\n");
379         return 1;
380     }
381 
382     *start = zi->zi_start;
383     *end = zi->zi_end;
384 
385     printf("Kernel image @ %#08lx [ %#08lx - %#08lx ]\n", image,
```

```
386             *start, *end);
387 
388     return 0;
389 }
```

第 370 行，宏 LINUX_ARM_ZIMAGE_MAGIC 就是 ARM Linux 系统魔术数。

第 376 行，从传递进来的参数 image(也就是系统镜像首地址)中获取 zimage 头。zImage 头结构体为 zimage_header。

第 377~380 行，判断 image 是否为 ARM 的 Linux 系统镜像，如果不是的话就直接返回，并且打印出“Bad Linux ARM zImage magic!”，比如我们输入一个错误的启动命令：

```
bootz 80000000 – 900000000
```

因为我们并没有在 0X80000000 处存放 Linux 镜像文件(zImage)，因此上面的命令肯定会执行出错的，结果如图 32.3.3.1 所示：

```
=> bootz 80000000 - 90000000
Bad Linux ARM zImage magic!
=>
```

图 32.3.3.1 启动出错

第 382、383 行初始化函数 bootz_setup 的参数 start 和 end。

第 385 行，打印启动信息，如果 Linux 系统镜像正常的话就会输出图 32.3.3.2 所示的信息：

```
Kernel image @ 0x80800000 [ 0x000000 - 0x65ef68 ]
```

图 32.3.3.3 Linux 镜像信息

接下来看一下函数 bootm_find_images，此函数定义在文件 common/bootm.c 中，函数内容如下：

示例代码 32.3.3.3 bootm_find_images 函数

```
225 int bootm_find_images(int flag, int argc, char * const argv[])
226 {
227     int ret;
228 
229     /* find ramdisk */
230     ret = boot_get_ramdisk(argc, argv, &images, IH_INITRD_ARCH,
231                 &images.rd_start, &images.rd_end);
232     if (ret) {
233         puts("Ramdisk image is corrupt or invalid\n");
234         return 1;
235     }
236 
237 #if defined(CONFIG_OF_LIBFDT)
238     /* find flattened device tree */
239     ret = boot_get_fdt(flag, argc, argv, IH_ARCH_DEFAULT, &images,
240              &images.ft_addr, &images.ft_len);
241     if (ret) {
242         puts("Could not find a valid device tree\n");
243         return 1;
```

```
244     }
245     set_working_fdt_addr((ulong)images.ft_addr);
246 #endif
......
258     return 0;
259 }
```

第 230~235 行是跟查找 ramdisk，但是我们没有用到 ramdisk，因此这部分代码不用管。

第 237~244 行是查找设备树(dtb)文件，找到以后就将设备树的起始地址和长度分别写到 images 的 ft_addr 和 ft_len 成员变量中。我们使用 bootz 启动 Linux 的时候已经指明了设备树在 DRAM 中的存储地址，因此 images.ft_addr=0X83000000，长度根据具体的设备树文件而定，比如我现在使用的设备树文件长度为 0X8C81，因此 images.ft_len=0X8C81。

bootz_start 函数就讲解到这里，bootz_start 主要用于初始化 images 的相关成员变量。

32.3.4 do_bootm_states 函数

do_bootz 最后调用的就是函数 do_bootm_states，而且在 bootz_start 中也调用了 do_bootm_states 函数，看来 do_bootm_states 函数还是个香饽饽。此函数定义在文件 common/bootm.c 中，函数代码如下：

示例代码 32.3.4.1 do_bootm_states 函数

```
591 int do_bootm_states(cmd_tbl_t *cmdtp, int flag, int argc, char *
const argv[],
592             int states, bootm_headers_t *images, int boot_progress)
593 {
594     boot_os_fn *boot_fn;
595     ulong iflag = 0;
596     int ret = 0, need_boot_fn;
597 
598     images->state |= states;
599 
600     /*
601      * Work through the states and see how far we get. We stop on
602      * any error.
603      */
604     if (states & BOOTM_STATE_START)
605         ret = bootm_start(cmdtp, flag, argc, argv);
606 
607     if (!ret && (states & BOOTM_STATE_FINDOS))
608         ret = bootm_find_os(cmdtp, flag, argc, argv);
609 
610     if (!ret && (states & BOOTM_STATE_FINDOTHER)) {
611         ret = bootm_find_other(cmdtp, flag, argc, argv);
612         argc = 0;   /* consume the args */
613     }
614 
```

```
615     /* Load the OS */
616     if (!ret && (states & BOOTM_STATE_LOADOS)) {
617         ulong load_end;
618 
619         iflag = bootm_disable_interrupts();
620         ret = bootm_load_os(images, &load_end, 0);
621         if (ret == 0)
622             lmb_reserve(&images->lmb, images->os.load,
623                     (load_end - images->os.load));
624         else if (ret && ret != BOOTM_ERR_OVERLAP)
625             goto err;
626         else if (ret == BOOTM_ERR_OVERLAP)
627             ret = 0;
628 #if defined(CONFIG_SILENT_CONSOLE)
&& !defined(CONFIG_SILENT_U_BOOT_ONLY)
629         if (images->os.os == IH_OS_LINUX)
630             fixup_silent_linux();
631 #endif
632     }
633 
634     /* Relocate the ramdisk */
635 #ifdef CONFIG_SYS_BOOT_RAMDISK_HIGH
636     if (!ret && (states & BOOTM_STATE_RAMDISK)) {
637         ulong rd_len = images->rd_end - images->rd_start;
638 
639         ret = boot_ramdisk_high(&images->lmb, images->rd_start,
640             rd_len, &images->initrd_start, &images->initrd_end);
641         if (!ret) {
642             setenv_hex("initrd_start", images->initrd_start);
643             setenv_hex("initrd_end", images->initrd_end);
644         }
645     }
646 #endif
647 #if defined(CONFIG_OF_LIBFDT) && defined(CONFIG_LMB)
648     if (!ret && (states & BOOTM_STATE_FDT)) {
649         boot_fdt_add_mem_rsv_regions(&images->lmb, images->ft_addr);
650         ret = boot_relocate_fdt(&images->lmb, &images->ft_addr,
651                     &images->ft_len);
652     }
653 #endif
654 
655     /* From now on, we need the OS boot function */
656     if (ret)
```

```
657         return ret;
658     boot_fn = bootm_os_get_boot_func(images->os.os);
659     need_boot_fn = states & (BOOTM_STATE_OS_CMDLINE |
660             BOOTM_STATE_OS_BD_T | BOOTM_STATE_OS_PREP |
661             BOOTM_STATE_OS_FAKE_GO | BOOTM_STATE_OS_GO);
662     if (boot_fn == NULL && need_boot_fn) {
663         if (iflag)
664             enable_interrupts();
665         printf("ERROR: booting os '%s' (%d) is not supported\n",
666                genimg_get_os_name(images->os.os), images->os.os);
667         bootstage_error(BOOTSTAGE_ID_CHECK_BOOT_OS);
668         return 1;
669     }
670 
671     /* Call various other states that are not generally used */
672     if (!ret && (states & BOOTM_STATE_OS_CMDLINE))
673         ret = boot_fn(BOOTM_STATE_OS_CMDLINE, argc, argv, images);
674     if (!ret && (states & BOOTM_STATE_OS_BD_T))
675         ret = boot_fn(BOOTM_STATE_OS_BD_T, argc, argv, images);
676     if (!ret && (states & BOOTM_STATE_OS_PREP))
677         ret = boot_fn(BOOTM_STATE_OS_PREP, argc, argv, images);
678 
679 #ifdef CONFIG_TRACE
680     /* Pretend to run the OS, then run a user command */
681     if (!ret && (states & BOOTM_STATE_OS_FAKE_GO)) {
682         char *cmd_list = getenv("fakegocmd");
683 
684         ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_FAKE_GO,
685                 images, boot_fn);
686         if (!ret && cmd_list)
687             ret = run_command_list(cmd_list, -1, flag);
688     }
689 #endif
690 
691     /* Check for unsupported subcommand. */
692     if (ret) {
693         puts("subcommand not supported\n");
694         return ret;
695     }
696 
697     /* Now run the OS! We hope this doesn't return */
698     if (!ret && (states & BOOTM_STATE_OS_GO))
699         ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_GO,
```

```
700                 images, boot_fn);
......
712     return ret;
713 }
```

函数 do_bootm_states 根据不同的 BOOT 状态执行不同的代码段，通过如下代码来判断 BOOT 状态：

```
states & BOOTM_STATE_XXX
```

在 do_bootz 函数中会用到 BOOTM_STATE_OS_PREP 、BOOTM_STATE_OS_FAKE_GO 和 BOOTM_STATE_OS_GO 这三个 BOOT 状态，bootz_start 函数中会用到 BOOTM_STATE_START 这个 BOOT 状态。为了精简代码，方便分析，因此我们将示例代码 32.3.4.1 中的函数 do_bootm_states 进行精简，只留下下面这 4 个 BOOT 状态对应的处理代码：

```
BOOTM_STATE_OS_PREP
BOOTM_STATE_OS_FAKE_GO
BOOTM_STATE_OS_GO
BOOTM_STATE_START
```

精简以后的 do_bootm_states 函数如下所示：

示例代码 32.3.4.2 精简后的 do_bootm_states 函数

```
591 int do_bootm_states(cmd_tbl_t *cmdtp, int flag, int argc, char *
const argv[],
592             int states, bootm_headers_t *images, int boot_progress)
593 {
594     boot_os_fn *boot_fn;
595     ulong iflag = 0;
596     int ret = 0, need_boot_fn;
597 
598     images->state |= states;
599 
600     /*
601      * Work through the states and see how far we get. We stop on
602      * any error.
603      */
604     if (states & BOOTM_STATE_START)
605         ret = bootm_start(cmdtp, flag, argc, argv);
......
654 
655     /* From now on, we need the OS boot function */
656     if (ret)
657         return ret;
658     boot_fn = bootm_os_get_boot_func(images->os.os);
659     need_boot_fn = states & (BOOTM_STATE_OS_CMDLINE |
660             BOOTM_STATE_OS_BD_T | BOOTM_STATE_OS_PREP |
661             BOOTM_STATE_OS_FAKE_GO | BOOTM_STATE_OS_GO);
662     if (boot_fn == NULL && need_boot_fn) {
```

```
663         if (iflag)
664             enable_interrupts();
665         printf("ERROR: booting os '%s' (%d) is not supported\n",
666                genimg_get_os_name(images->os.os), images->os.os);
667         bootstage_error(BOOTSTAGE_ID_CHECK_BOOT_OS);
668         return 1;
669     }
670 
......
676     if (!ret && (states & BOOTM_STATE_OS_PREP))
677         ret = boot_fn(BOOTM_STATE_OS_PREP, argc, argv, images);
678 
679 #ifdef CONFIG_TRACE
680     /* Pretend to run the OS, then run a user command */
681     if (!ret && (states & BOOTM_STATE_OS_FAKE_GO)) {
682         char *cmd_list = getenv("fakegocmd");
683 
684         ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_FAKE_GO,
685                 images, boot_fn);
686         if (!ret && cmd_list)
687             ret = run_command_list(cmd_list, -1, flag);
688     }
689 #endif
690 
691     /* Check for unsupported subcommand. */
692     if (ret) {
693         puts("subcommand not supported\n");
694         return ret;
695     }
696 
697     /* Now run the OS! We hope this doesn't return */
698     if (!ret && (states & BOOTM_STATE_OS_GO))
699         ret = boot_selected_os(argc, argv, BOOTM_STATE_OS_GO,
700                 images, boot_fn);
......
712     return ret;
713 }
```

第 604、605 行，处理 BOOTM_STATE_START 阶段，bootz_start 会执行这一段代码，这里调用函数 bootm_start，此函数定义在文件 common/bootm.c 中，函数内容如下：

示例代码 32.3.4.2 bootm_start 函数

```
69 static int bootm_start(cmd_tbl_t *cmdtp, int flag, int argc,
70             char * const argv[])
71 {
```

```
72      memset((void *)&images, 0, sizeof(images)); /* 清空 images */
73      images.verify = getenv_yesno("verify");/* 初始化 verfify 成员 */
74 
75      boot_start_lmb(&images);
76 
77      bootstage_mark_name(BOOTSTAGE_ID_BOOTM_START, "bootm_start");
78      images.state = BOOTM_STATE_START;/* 设置状态为 BOOTM_STATE_START */
79 
80      return 0;
81 }
```

接着回到示例代码 32.3.4.2 中，继续分析函数 do_bootm_states。第 658 行非常重要！通过函数 bootm_os_get_boot_func 来查找系统启动函数，参数 images->os.os 就是系统类型，根据这个系统类型来选择对应的启动函数，在 do_bootz 中设置 images.os.os= IH_OS_LINUX。函数返回值就是找到的系统启动函数，这里找到的 Linux 系统启动函数为 do_bootm_linux，关于此函数查找系统启动函数的过程请参考 32.3.5 小节。因此 boot_fn=do_bootm_linux，后面执行 boot_fn 函数的地方实际上是执行的 do_bootm_linux 函数。

第 676 行，处理 BOOTM_STATE_OS_PREP 状态，调用函数 do_bootm_linux，do_bootm_linux 也是调用 boot_prep_linux 来完成具体的处理过程。boot_prep_linux 主要用于处理环境变量 bootargs，bootargs 保存着传递给 Linux kernel 的参数。

第 679~689 行是处理 BOOTM_STATE_OS_FAKE_GO 状态的，但是要我们没用使能 TRACE 功能，因此宏 CONFIG_TRACE 也就没有定义，所以这段程序不会编译。

第 699 行，调用函数 boot_selected_os 启动 Linux 内核，此函数第 4 个参数为 Linux 系统镜像头，第 5 个参数就是 Linux 系统启动函数 do_bootm_linux。boot_selected_os 函数定义在文件 common/bootm_os.c 中，函数内容如下：

示例代码 32.3.4.3 boot_selected_os 函数

```
476 int boot_selected_os(int argc, char * const argv[], int state,
477                 bootm_headers_t *images, boot_os_fn *boot_fn)
478 {
479     arch_preboot_os();
480     boot_fn(state, argc, argv, images);
......
490     return BOOTM_ERR_RESET;
491 }
```

第 480 行调用 boot_fn 函数，也就是 do_bootm_linux 函数来启动 Linux 内核。

32.3.5 bootm_os_get_boot_func 函数

do_bootm_states 会调用 bootm_os_get_boot_func 来查找对应系统的启动函数，此函数定义在文件 common/bootm_os.c 中，函数内容如下：

示例代码 32.3.5.1 bootm_os_get_boot_func 函数

```
493 boot_os_fn *bootm_os_get_boot_func(int os)
494 {
495 #ifdef CONFIG_NEEDS_MANUAL_RELOC
496     static bool relocated;
```

```
497
498     if (!relocated) {
499         int i;
500
501         /* relocate boot function table */
502         for (i = 0; i < ARRAY_SIZE(boot_os); i++)
503             if (boot_os[i] != NULL)
504                 boot_os[i] += gd->reloc_off;
505
506         relocated = true;
507     }
508 #endif
509     return boot_os[os];
510 }
```

第 495~508 行是条件编译，在本 uboot 中没有用到，因此这段代码无效，只有 509 行有效。在 509 行中 boot_os 是个数组，这个数组里面存放着不同的系统对应的启动函数。boot_os 也定义在文件 common/bootm_os.c 中，如下所示：

示例代码 32.3.5.2 boot_os 数组

```
435 static boot_os_fn *boot_os[] = {
436     [IH_OS_U_BOOT] = do_bootm_standalone,
437 #ifdef CONFIG_BOOTM_LINUX
438     [IH_OS_LINUX] = do_bootm_linux,
439 #endif
......
465 #ifdef CONFIG_BOOTM_OPENRTOS
466     [IH_OS_OPENRTOS] = do_bootm_openrtos,
467 #endif
468 };
```

第 438 行就是 Linux 系统对应的启动函数：do_bootm_linux。

32.3.6 do_bootm_linux 函数

经过前面的分析，我们知道了 do_bootm_linux 就是最终启动 Linux 内核的函数，此函数定义在文件 arch/arm/lib/bootm.c，函数内容如下：

示例代码 32.3.6.1 do_bootm_linux 函数

```
339 int do_bootm_linux(int flag, int argc, char * const argv[],
340             bootm_headers_t *images)
341 {
342     /* No need for those on ARM */
343     if (flag & BOOTM_STATE_OS_BD_T || flag & BOOTM_STATE_OS_CMDLINE)
344         return -1;
345
346     if (flag & BOOTM_STATE_OS_PREP) {
347         boot_prep_linux(images);
```

```
348         return 0;
349     }
350 
351     if (flag & (BOOTM_STATE_OS_GO | BOOTM_STATE_OS_FAKE_GO)) {
352         boot_jump_linux(images, flag);
353         return 0;
354     }
355 
356     boot_prep_linux(images);
357     boot_jump_linux(images, flag);
358     return 0;
359 }
```

第 351 行，如果参数 flag 等于 BOOTM_STATE_OS_GO 或者 BOOTM_STATE_OS_FAKE_GO 的话就执行 boot_jump_linux 函数。boot_selected_os 函数在调用 do_bootm_linux 的时候会将 flag 设置为 BOOTM_STATE_OS_GO。

第 352 行，执行函数 boot_jump_linux(又来了一个函数，绕啊绕啊！心累！)，此函数定义在文件 arch/arm/lib/bootm.c 中，函数内容如下：

示例代码 32.3.6.2 boot_jump_linux 函数

```
272 static void boot_jump_linux(bootm_headers_t *images, int flag)
273 {
274 #ifdef CONFIG_ARM64
......
292 #else
293     unsigned long machid = gd->bd->bi_arch_number;
294     char *s;
295     void (*kernel_entry)(int zero, int arch, uint params);
296     unsigned long r2;
297     int fake = (flag & BOOTM_STATE_OS_FAKE_GO);
298 
299     kernel_entry = (void (*)(int, int, uint))images->ep;
300 
301     s = getenv("machid");
302     if (s) {
303         if (strict_strtoul(s, 16, &machid) < 0) {
304             debug("strict_strtoul failed!\n");
305             return;
306         }
307         printf("Using machid 0x%lx from environment\n", machid);
308     }
309 
310     debug("## Transferring control to Linux (at address %08lx)" \
311         "...\n", (ulong) kernel_entry);
312     bootstage_mark(BOOTSTAGE_ID_RUN_OS);
```

```
313     announce_and_cleanup(fake);
314 
315     if (IMAGE_ENABLE_OF_LIBFDT && images->ft_len)
316         r2 = (unsigned long)images->ft_addr;
317     else
318         r2 = gd->bd->bi_boot_params;
319 
......
328             kernel_entry(0, machid, r2);
329     }
330 #endif
331 }
```

第 274~292 行是 64 位 ARM 芯片对应的代码，Cortex-A7 是 32 位芯片，因此用不到。

第 293 行，变量 machid 保存机器 ID，如果不使用设备树的话这个机器 ID 会被传递给 Linux 内核，Linux 内核会在自己的机器 ID 列表里面查找是否存在与 uboot 传递进来的 machid 匹配的项目，如果存在就说 Linux 内核支持这个机器，那么 Linux 就会启动！如果使用设备树的话这个 machid 就无效了，设备树存有一个"兼容性"这个属性，Linux 内核会比较"兼容性"属性的值(字符串)来查看是否支持这个机器。

第 295 行，函数 kernel_entry，看名字"内核_进入"，说明此函数是进入 Linux 内核的，也就是最终的大 boos!! 此函数有三个参数：zero，arch，params，第一个参数 zero 同样为 0；第二个参数为机器 ID；第三个参数 ATAGS 或者设备树(DTB)首地址，ATAGS 是传统的方法，用于传递一些命令行信息啥的，如果使用设备树的话就要传递设备树(DTB)。

第 299 行，获取 kernel_entry 函数，函数 kernel_entry 并不是 uboot 定义的，而是 Linux 内核定义的，Linux 内核镜像文件的第一行代码就是函数 kernel_entry，而 images->ep 保存着 Linux 内核镜像的起始地址，而起始地址保存的不正是 Linux 内核第一行代码！

第 313 行，调用函数 announce_and_cleanup 来打印一些信息并做一些清理工作，此函数定义在文件 arch/arm/lib/bootm.c 中，函数内容如下：

示例代码 32.3.6.3 announce_and_cleanup 函数

```
72 static void announce_and_cleanup(int fake)
73 {
74     printf("\nStarting kernel ...%s\n\n", fake ?
75         "(fake run for tracing)" : "");
76     bootstage_mark_name(BOOTSTAGE_ID_BOOTM_HANDOFF, "start_kernel");
......
87     cleanup_before_linux();
88 }
```

第 74 行，在启动 Linux 之前输出"Starting kernel ..."信息，如图 32.3.6.1 所示：

```
Kernel image @ 0x80800000 [ 0x000000 - 0x65ef68 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300bc80

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.
t May 25 12:32:15 CST 2019
```

图 32.3.6.1 系统启动提示信息

第 87 行调用 cleanup_before_linux 函数做一些清理工作。

继续回到示例代码 32.3.6.2 的函数 boot_jump_linux，第 315~318 行是设置寄存器 r2 的值？为什么要设置 r2 的值呢？Linux 内核一开始是汇编代码，因此函数 kernel_entry 就是个汇编函数。向汇编函数传递参数要使用 r0、r1 和 r2(参数数量不超过 3 个的时候)，所以 r2 寄存器就是函数 kernel_entry 的第三个参数。

第 316 行，如果使用设备树的话，r2 应该是设备树的起始地址，而设备树地址保存在 images 的 ftd_addr 成员变量中。

第 317 行，如果不使用设备树的话，r2 应该是 uboot 传递给 Linux 的参数起始地址，也就是环境变量 bootargs 的值，

第 328 行，调用 kernel_entry 函数进入 Linux 内核，此行将一去不复返，uboot 的使命也就完成了，它可以安息了！

总结一下 bootz 命令的执行过程，如图 32.3.6.2 所示：

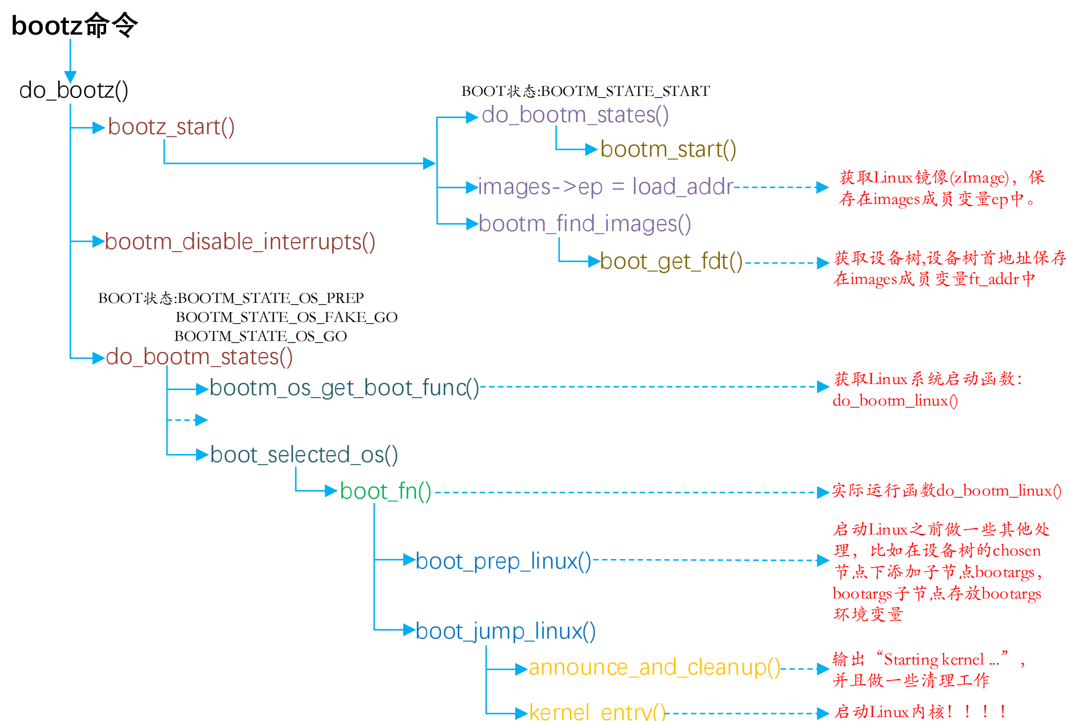


图 32.3.6.2 bootz 命令执行过程

到这里 uboot 的启动流程我们就讲解完成了，加上 uboot 顶层 Makefile 的分析，洋洋洒洒 100 多页，还是不少的！这也仅仅是 uboot 启动流程分析，当缕清了 uboot 的启动流程以后，后面移植 uboot 就会轻松很多。其实在工作中我们基本不需要这么详细的去了解 uboot，半导体厂商提供给我们的 uboot 一般是可以直接用的，只要能跑起来，可以使用就可以了。但是作为学习，我们是必须去详细的了解一下 uboot 的启动流程，否则如果在工作中遇到问题我们连解决的方法都没有，都不知道该从哪里看起。但是呢，如果第一次就想弄懂 uboot 的整个启动流程还是有点困难的，所以如果没有看懂的话，不要紧！不要气馁，大多数人第一次看 uboot 启动流程基本都有各种各样的问题。

题外话：

相信大家看完本章以后基本都有一个感觉：长、复杂、绕！没错，当我第一次学习 uboot 的时候看到 uboot 启动流程的时候也是这个感觉，当时我也一脸懵逼，怎么这么复杂，这么长呢？

尤其前面的汇编代码部分，还要涉及到 ARM 处理器架构的内容，当时也怀疑过自己是不是搞这一块的料。不过好在自己坚持下来了，uboot 的启动流程我至少分析过 7,8 遍，各种版本的，零几年很古老的；12 年、14 年比较新的等等很多个版本的 uboot。就 I.MX6ULL 使用的这个 2016.03 版本 uboot 我至少详细的分析了 2 遍，直至写完本章，大概花了 1 个月的时间。这期间查阅了各种资料，看了不知道多少篇博客，在这里感谢那些无私奉献的网友们。

相信很多朋友看完本章可能会想：我什么时候也能这么厉害，能够这么详细的分析 uboot 启动流程。甚至可能会有挫败感，还是那句话：不要气馁！千里之行始于足下，所有你羡慕的人都曾经痛苦过，挫败过。脚踏实地，一步一个脚印，一点一滴的积累，最终你也会成为你所羡慕的人。在嵌入式 Linux 这条道路上，有众多的学习者陪着你，大家相互搀扶，终能踏出一条康庄大道，祝所有的同学终有所获！

第三十三章 U-Boot 移植

上一章节我们详细的分析了 uboot 的启动流程，对 uboot 有了一个初步的了解。前两章我们都是使用的正点原子提供的 uboot，本章我们就来学习如何将 NXP 官方的 uboot 移植到正点原子的 I.MX6ULL 开发板上，学习如何在 uboot 中添加我们自己的板子。

33.1 NXP 官方开发板 uboot 编译测试

33.1.1 查找 NXP 官方的开发板默认配置文件

uboot 的移植并不是说我们完完全全的从零开始将 uboot 移植到我们现在所使用的开发板或者开发平台上。这个对于我们来说基本是不可能的，这个工作一般是半导体厂商做的，半导体厂商负责将 uboot 移植到他们的芯片上，因此半导体厂商都会自己做一个开发板，这个开发板就叫做原厂开发板，比如大家学习 STM32 的时候听说过的 discover 开发板就是 ST 自己做的。半导体厂商会将 uboot 移植到他们自己的原厂开发板上，测试好以后就会将这个 uboot 发布出去，这就是大家常说的原厂 BSP 包。我们一般做产品的时候就会参考原厂的开发板做硬件，然后在原厂提供的 BSP 包上做修改，将 uboot 或者 linux kernel 移植到我们的硬件上。这个就是 uboot 移植的一般流程：

①、在 uboot 中找到参考的开发平台，一般是原厂的开发板。

②、参考原厂开发板移植 uboot 到我们所使用的开发板上。

正点原子的 I.MX6ULL 开发板参考的是 NXP 官方的 I.MX6ULL EVK 开发板做的硬件，因此我们在移植 uboot 的时候就可以以 NXP 官方的 I.MX6ULL EVK 开发板为蓝本。

本章我们是将 NXP 官方的 uboot 移植到正点原子的 I.MX6ULL 开发板上，NXP 官方的 uboot 放到了开发板光盘中，路径为：1、例程源码->4、NXP 官方原版 Uboot 和 Linux->uboot-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2。将 uboot-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2 发送到 Ubuntu 中并解压，然后创建 VSCode 工程。

在移植之前，我们先编译一下 NXP 官方 I.MX6ULL EVK 开发板对应的 uboot，首先是配置 uboot，configs 目录下有很多跟 I.MX6UL/6ULL 有关的配置如图 33.1.1.1 所示，

IX 6UL > IMX6UL NXP UBOOT And Linux > IMX6UL > uboot > uboot-imx-rel_imx_4.1.15_2.1.0_ga > configs

名称	修改日期	类型	大小
mx6ul_9x9_evk_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_eimnor_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_emmc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_nand_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_ddr3_arm2_spinor_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_ddr_eol_brillo_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_ddr_eol_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_ddr_eol_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_emmc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_nand_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_evk_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_lpddr2_arm2_defconfig	2017/5/2 10:45	文件	1 KB
mx6ul_14x14_lpddr2_arm2_eimnor_defconf...	2017/5/2 10:45	文件	1 KB
mx6ull_9x9_evk_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_9x9_evk_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_emmc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_epdc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_nand_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_qspi1_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_spinor_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_ddr3_arm2_tsc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_evk_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_evk_emmc_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_evk_nand_defconfig	2017/5/2 10:45	文件	1 KB
mx6ull_14x14_evk_qspi1_defconfig	2017/5/2 10:45	文件	1 KB

图 33.1.1.1 NXP 官方 I.MX6UL/6ULL 默认配置文件

从图 33.1.1.1 可以看出有很多的默认配置文件，其中以 mx6ul 开头的是 I.MX6UL 芯片的，mx6ull 开头的是 I.MX6ULL 开发板的。I.MX6UL/6ULL 有 9x9mm 和 14x14mm 两种尺寸的，所以我们可以看到会有 mx6ull_9x9 和 mx6ull_14x14 开头的默认配置文件。我们使用的是 14x14mm 的芯片，所以关注 mx6ull_14x14 开头的默认配置文件。正点原子的 I.MX6ULL 有 EMMC 和 NAND 两个版本的，因此我们最终只需要关注 mx6ull_14x14_evk_emmc_defconfig 和 mx6ull_14x14_evk_nand_defconfig 这两个配置文件就行了。本章我们讲解 EMMC 版本的移植（NAND 版本移植很多类似），所以使用 mx6ull_14x14_evk_emmc_defconfig 作为默认配置文件。

33.1.2 编译 NXP 官方开发板对应的 uboot

找到 NXP 官方 I.MX6ULL EVK 开发板对应的默认配置文件以后就可以编译一下，使用如下命令编译 uboot：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_14x14_evk_emmc_defconfig
make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j16
```

编译完成以后结果如图 33.1.2.1 所示：

```
  arm-linux-gnueabihf-objcopy --gap-fill=0xff  -j .text -j .secure_text -j .rodata -j .hash
-j .data -j .got -j .got.plt -j .u_boot_list -j .rel.dyn -O binary  u-boot u-boot-nodtb.bin
  arm-linux-gnueabihf-objcopy --gap-fill=0xff  -j .text -j .secure_text -j .rodata -j .hash
-j .data -j .got -j .got.plt -j .u_boot_list -j .rel.dyn -O srec u-boot u-boot.srec
  arm-linux-gnueabihf-objdump -t u-boot > u-boot.sym
  cp u-boot-nodtb.bin u-boot.bin
make -f ./scripts/Makefile.build obj=arch/arm/imx-common u-boot.imx
mkdir -p board/freescale/mx6ullevk/
  ./tools/mkimage -n board/freescale/mx6ullevk/imximage.cfg.cfgtmp -T imximage -e 0x87800000
 -d u-boot.bin u-boot.imx
Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6/7 compatible)
Mode:         DCD
Data Size:    425984 Bytes = 416.00 kB = 0.41 MB
Load Address: 877ff420
Entry Point:  87800000
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/temp/uboot-imx-rel_imx_4.1.15_2.1.0_ga$
```

图 33.1.2.1 编译结果

从图 33.1.2.1 可以看出，编译成功。我们在编译的时候需要输入 ARCH 和 CORSS_COMPILE 这两个变量的值，这样太麻烦了。我们可以直接在顶层 Makefile 中直接给 ARCH 和 CORSS_COMPILE 赋值，修改如图 33.1.2.2 所示：

```
245  # set default to nothing for native builds
246  ifeq ($(HOSTARCH),$(ARCH))
247  CROSS_COMPILE ?=
248  endif
249
250  ARCH = arm
251  CROSS_COMPILE = arm-linux-gnueabihf-
```

图 33.1.2.2 添加 ARCH 和 CROSS_COMPILE 值

图 33.1.2.2 中的 250、251 行就是直接给 ARCH 和 CROSS_COMPILE 赋值，这样我们就可以使用如下简短的命令来编译 uboot 了：

```
make mx6ull_14x14_evk_emmc_defconfig
make V=1 -j16
```

如果既不想修改 uboot 的顶层 Makefile，又想编译的时候不用输入那么多，那么就直接创建个 shell 脚本就行了，shell 脚本名为 mx6ull_14x14_emmc.sh，然后在 shell 脚本里面输入如下内容：

示例代码 33.1.2.1 mx6ull_14x14_emmc.sh 文件

```
1 #!/bin/bash
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
mx6ull_14x14_evk_emmc_defconfig
4 make V=1 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j16
```

记得给 mx6ull_14x14_emmc.sh 这个文件可执行权限，使用 mx6ull_14x14_emmc.sh 脚本编译 uboot 的时候每次都会清理一下工程，然后全部重新编译，编译的时候直接执行这个脚本就行了，命令如下：

```
./mx6ull_14x14_evk_emmc.sh
```

编译完成以后会生成 u-boot.bin、u-boot.imx 等文件，但是这些文件是 NXP 官方 I.MX6ULL EVK 开饭。能不能用到正点原子的 I.MX6ULL 开发板上呢？试一下不就知道了！

33.1.3 烧写验证与驱动测试

将 imxdownload 软件拷贝到 uboot 源码根目录下，然后使用 imxdownload 软件将 u-boot.bin 烧写到 SD 卡中，烧写命令如下：

```
chmod 777 imxdownload                    //给予 imxdownload 可执行权限
./imxdownload u-boot.bin /dev/sdg        //烧写 u-boot.bin 到 SD 卡中
```

烧写完成以后将 SD 卡插入 I.MX6U-ALPHA 开发板的 TF 卡槽中，最后设置开发板从 SD 卡启动。打开 SecureCRT，设置好开发板所使用的串口并打开，复位开发板，SecureCRT 接收到如下图 33.1.3.1 所示信息：

```
U-Boot 2016.03 (May 11 2019 - 15:52:17 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 45C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
unsupported panel TFT7016
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
Net:   Board Net Initialization Failed
No ethernet found.
Normal Boot
Hit any key to stop autoboot:  0
No ethernet found.
No ethernet found.
Bad Linux ARM zImage magic!
=>
```

图 33.1.3.1 uboot 启动信息

从图 33.1.3.1 可以看出，uboot 启动正常，虽然我们用的是 NXP 官方 I.MX6ULL 开发板的 uboot，但是在正点原子的 I.MX6ULL 开发板上是可以正常启动的。而且 DRAM 识别正确，为 512MB，如果用的 NAND 版本的核心版的话 uboot 启动会失败！因为 NAND 核心版用的 256MB 的 DRAM。

1、SD 卡和 EMMC 驱动检查

检查一下 SD 卡和 EMMC 驱动是否正常，使用命令 mmc list 列出当前的 MMC 设备，结果如图 33.1.3.2 所示：

```
=> mmc list
FSL_SDHC: 0 (SD)
FSL_SDHC: 1
=>
```

图 33.1.3.2 emmc 设备检查

从图 33.1.3.2 可以看出当前有两个 MMC 设备，检查每个 MMC 设备信息，先检查 MMC 设备 0，输入如下命令：

```
mmc dev 0
mmc info
```

结果如图 33.1.3.3 所示：

```
=> mmc dev 0
switch to partitions #0, OK
mmc0 is current device
=> mmc info
Device: FSL_SDHC
Manufacturer ID: 3
OEM: 5344
Name: SC16G
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
```

图 33.1.3.3 mmc 设备 0 信息

从图 33.1.3.3 可以看出，mmc 设备 0 是 SD 卡，SD 卡容量为 14.8GB，这个和我所使用的 SD 卡信息相符，说明 SD 卡驱动正常。再来检查 MMC 设备 1，输入如下命令：

```
mmc dev 1
mmc info
```

结果如图 33.1.3.4 所示：

```
=> mmc dev 1
switch to partitions #0, OK
mmc1(part 0) is current device
=> mmc info
Device: FSL_SDHC
Manufacturer ID: 15
OEM: 100
Name: 4FTE4
Tran Speed: 52000000
Rd Block Len: 512
MMC version 4.0
High Capacity: Yes
Capacity: 3.6 GiB
Bus Width: 8-bit
Erase Group Size: 512 KiB
=>
```

图 33.1.3.4 mmc 设备 1 信息

从图 33.1.3.4 可以看出，mmc 设备 1 为 EMMC，容量为 3.6GB，说明 EMMC 驱动也成功，SD 卡和 EMMC 的驱动都没问题。

2、LCD 驱动检查

如果 uboot 中的 LCD 驱动正确的话，启动 uboot 以后 LCD 上应该会显示出 NXP 的 logo，如下图 33.1.3.5 所示：

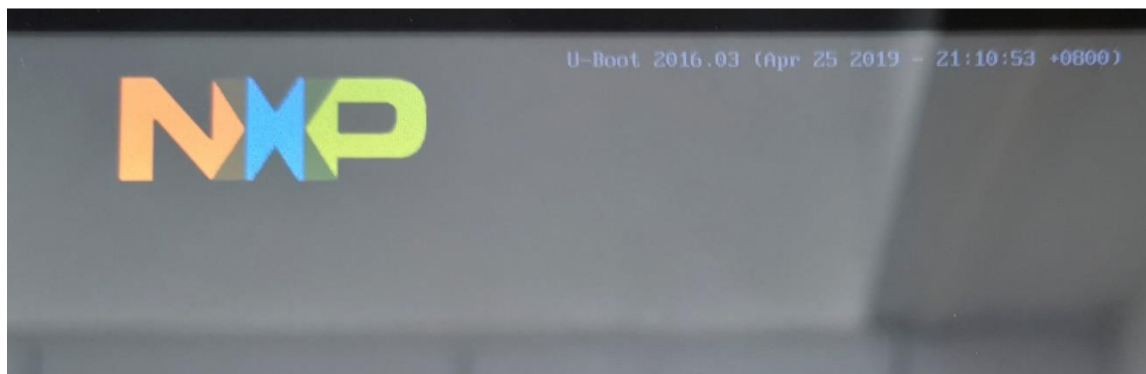


图 33.1.3.5 uboot LCD 界面

如果你用的不是正点原子的 4.3 寸 480x272 分辨率的屏幕的话，那么 LCD 就不会显示 33.1.3.5 所示 logo 界面。因为 NXP 官方 I.MX6ULL 开发板的屏幕就是 4.3 寸 480x272 分辨率的，所以 uboot 默认 LCD 驱动是 4.3 寸 480x272 分辨率的。如果使用其他分辨率的 LCD 就需要修改 LCD 驱动，这里我们先不修改 LCD 驱动了，稍后我们在讲解如何修改 uboot 中的 LCD 驱动，我们只需要记得，uboot 的 LCD 需要修改就行了。

3、网络驱动

uboot 启动的时候提示"Board Net Initialization Failed"和"No ethernet found."这两行，说明网络驱动也有问题，正常情况下应该是如图 33.1.3.6 所示提示：

```
switch to partitions #0, OK
mmc0 is current device
Net:   FEC0
Normal Boot
Hit any key to stop autoboot:  0
=>
```

图 33.1.3.6 网络信息

现在没有图 33.1.3.6 中的信息，那更别说 ping 一下 ubuntu 主机了，说明当前 uboot 的网络部驱动也是有问题的，这是因为正点原子开发板的网络芯片复位引脚和 NXP 官方开发板不一样，因此需要修改驱动。

总结一下 NXP 官方 I.MX6ULL EVK 开发板的 uboot 在正点原子 EMMC 版本 I.MX6ULL 开发板上的运行情况：

①、uboot 启动正常，DRAM 识别正确，SD 卡和 EMMC 驱动正常。

②、uboot 里面的 LCD 驱动默认是给 4.3 寸 480x272 分辨率的，如果使用的其他分辨率的屏幕需要修改驱动。

②、网络不能工作，识别不出来网络信息，需要修改驱动。

接下来我们要做的工作如下：

①、前面我们一直使用着 uboot 中 NXP 官方开发板的配置，接下来需要在 uboot 中添加我们自己的开发板，也就是正点原子的 I.MX6ULL 开发板。

②、解决 LCD 驱动和网络驱动的问题。

33.2 在 U-Boot 中添加自己的开发板

NXP 官方 uboot 中默认都是 NXP 自己的开发板，虽说我们可以直接在官方的开发板上直接修改，使 uboot 可以完整的运行在我们的板子上。但是从学习的角度来讲，这样我们就不能

了解到 uboot 是如何添加新平台的。接下来我们就参考 NXP 官方的 I.MX6ULL EVK 开发板，学习如何在 uboot 中添加我们的开发板或者开发平台。

33.2.1 添加开发板默认配置文件

先在 configs 目录下创建默认配置文件，复制 mx6ull_14x14_evk_emmc_defconfig，然后重命名为 mx6ull_alientek_emmc_defconfig，命令如下：

```
cd configs
cp mx6ull_14x14_evk_emmc_defconfig  mx6ull_alientek_emmc_defconfig
```

然后将文件 mx6ull_alientek_emmc_defconfig 中的内容改成下面的：

示例代码 33.2.1.1 mx6ull_alientek_emmc_defconfig 文件

```
1 CONFIG_SYS_EXTRA_OPTIONS="IMX_CONFIG=board/freescale/mx6ull_alientek_
  emmc/imximage.cfg,MX6ULL_EVK_EMMC_REWORK"
2 CONFIG_ARM=y
3 CONFIG_ARCH_MX6=y
4 CONFIG_TARGET_MX6ULL_ALIENTEK_EMMC=y
5 CONFIG_CMD_GPIO=y
```

可以看出，mx6ull_alientek_emmc_defconfig 基本和 mx6ull_14x14_evk_emmc_defconfig 中的内容一样，只是第 1 行和第 4 行做了修改。

33.2.2 添加开发板对应的头文件

在目录 include/configs 下添加 I.MX6ULL-ALPHA 开发板对应的头文件，复制 include/configs/mx6ullevk.h，并重命名为 mx6ull_alientek_emmc.h，命令如下：

```
cp include/configs/mx6ullevk.h mx6ull_alientek_emmc.h
```

拷贝完成以后将：

```
#ifndef __MX6ULLEVK_CONFIG_H
#define __MX6ULLEVK_CONFIG_H
```

改为：

```
#ifndef __MX6ULL_ALIENTEK_EMMC_CONFIG_H
#define __MX6ULL_ALIENTEK_EMMC_CONFIG_H
```

mx6ull_alientek_emmc.h 里面有很多宏定义，这些宏定义基本用于配置 uboot，也有一些 I.MX6ULL 的配置项目。如果我们自己要想使能或者禁止 uboot 的某些功能，那就在 mx6ull_alientek_emmc.h 里面做修改即可。mx6ull_alientek_emmc.h 里面的内容比较多，去掉一些用不到的配置，精简后的内容如下：

示例代码 33.2.2.1 mx6ull_alientek_emmc.h 文件

```
1  /*
2   * Copyright (C) 2016 Freescale Semiconductor, Inc.
3   *
4   * Configuration settings for the Freescale i.MX6UL 14x14 EVK board.
5   *
6   * SPDX-License-Identifier: GPL-2.0+
7   */
8  #ifndef __MX6ULL_ALEITENK_EMMC_CONFIG_H
9  #define __MX6ULL_ALEITENK_EMMC_CONFIG_H
```

```
10
11
12 #include <asm/arch/imx-regs.h>
13 #include <linux/sizes.h>
14 #include "mx6_common.h"
15 #include <asm/imx-common/gpio.h>
16
......
28
29 #define is_mx6ull_9x9_evk() CONFIG_IS_ENABLED(TARGET_MX6ULL_9X9_EVK)
30
31 #ifdef CONFIG_TARGET_MX6ULL_9X9_EVK
32 #define PHYS_SDRAM_SIZE     SZ_256M
33 #define CONFIG_BOOTARGS_CMA_SIZE   "cma=96M "
34 #else
35 #define PHYS_SDRAM_SIZE     SZ_512M
36 #define CONFIG_BOOTARGS_CMA_SIZE   ""
37 /* DCDC used on 14x14 EVK, no PMIC */
38 #undef CONFIG_LDO_BYPASS_CHECK
39 #endif
40
41 /* SPL options */
42 /* We default not support SPL
43  * #define CONFIG_SPL_LIBCOMMON_SUPPORT
44  * #define CONFIG_SPL_MMC_SUPPORT
45  * #include "imx6_spl.h"
46 */
47
48 #define CONFIG_ENV_VARS_UBOOT_RUNTIME_CONFIG
49
50 #define CONFIG_DISPLAY_CPUINFO
51 #define CONFIG_DISPLAY_BOARDINFO
52
53 /* Size of malloc() pool */
54 #define CONFIG_SYS_MALLOC_LEN      (16 * SZ_1M)
55
56 #define CONFIG_BOARD_EARLY_INIT_F
57 #define CONFIG_BOARD_LATE_INIT
58
59 #define CONFIG_MXC_UART
60 #define CONFIG_MXC_UART_BASE       UART1_BASE
61
62 /* MMC Configs */
```

```
63 #ifdef CONFIG_FSL_USDHC
64 #define CONFIG_SYS_FSL_ESDHC_ADDR   USDHC2_BASE_ADDR
65
66 /* NAND pin conflicts with usdhc2 */
67 #ifdef CONFIG_SYS_USE_NAND
68 #define CONFIG_SYS_FSL_USDHC_NUM    1
69 #else
70 #define CONFIG_SYS_FSL_USDHC_NUM    2
71 #endif
72 #endif
73
74 /* I2C configs */
75 #define CONFIG_CMD_I2C
76 #ifdef CONFIG_CMD_I2C
77 #define CONFIG_SYS_I2C
78 #define CONFIG_SYS_I2C_MXC
79 #define CONFIG_SYS_I2C_MXC_I2C1     /* enable I2C bus 1 */
80 #define CONFIG_SYS_I2C_MXC_I2C2     /* enable I2C bus 2 */
81 #define CONFIG_SYS_I2C_SPEED       100000
82
......
89
90 #define CONFIG_SYS_MMC_IMG_LOAD_PART    1
91
92 #ifdef CONFIG_SYS_BOOT_NAND
93 #define CONFIG_MFG_NAND_PARTITION "mtdparts=gpmi-
nand:64m(boot),16m(kernel),16m(dtb),1m(misc),-(rootfs) "
94 #else
95 #define CONFIG_MFG_NAND_PARTITION ""
96 #endif
97
98 #define CONFIG_MFG_ENV_SETTINGS \
99     "mfgtool_args=setenv bootargs console=${console},${baudrate} " \
......
111     "bootcmd_mfg=run mfgtool_args;bootz ${loadaddr} ${initrd_addr}
${fdt_addr};\0" \
112
113 #if defined(CONFIG_SYS_BOOT_NAND)
114 #define CONFIG_EXTRA_ENV_SETTINGS \
115     CONFIG_MFG_ENV_SETTINGS \
116     "panel=TFT43AB\0" \
......
126        "bootz ${loadaddr} - ${fdt_addr}\0"
```

```
127
128 #else
129 #define CONFIG_EXTRA_ENV_SETTINGS \
130     CONFIG_MFG_ENV_SETTINGS \
131     "script=boot.scr\0" \
......
202            "fi;\0" \
203
204 #define CONFIG_BOOTCOMMAND \
205        "run findfdt;" \
......
216        "else run netboot; fi"
217 #endif
218
219 /* Miscellaneous configurable options */
220 #define CONFIG_CMD_MEMTEST
221 #define CONFIG_SYS_MEMTEST_START   0x80000000
222 #define CONFIG_SYS_MEMTEST_END     (CONFIG_SYS_MEMTEST_START +
0x8000000)
223
224 #define CONFIG_SYS_LOAD_ADDR       CONFIG_LOADADDR
225 #define CONFIG_SYS_HZ              1000
226
227 #define CONFIG_STACKSIZE          SZ_128K
228
229 /* Physical Memory Map */
230 #define CONFIG_NR_DRAM_BANKS       1
231 #define PHYS_SDRAM                 MMDC0_ARB_BASE_ADDR
232
233 #define CONFIG_SYS_SDRAM_BASE      PHYS_SDRAM
234 #define CONFIG_SYS_INIT_RAM_ADDR   IRAM_BASE_ADDR
235 #define CONFIG_SYS_INIT_RAM_SIZE   IRAM_SIZE
236
237 #define CONFIG_SYS_INIT_SP_OFFSET \
238     (CONFIG_SYS_INIT_RAM_SIZE - GENERATED_GBL_DATA_SIZE)
239 #define CONFIG_SYS_INIT_SP_ADDR \
240     (CONFIG_SYS_INIT_RAM_ADDR + CONFIG_SYS_INIT_SP_OFFSET)
241
242 /* FLASH and environment organization */
243 #define CONFIG_SYS_NO_FLASH
244
......
255
```

```
256 #define CONFIG_SYS_MMC_ENV_DEV      1   /* USDHC2 */
257 #define CONFIG_SYS_MMC_ENV_PART     0   /* user area */
258 #define CONFIG_MMCROOT          "/dev/mmcblk1p2"  /* USDHC2 */
259
260 #define CONFIG_CMD_BMODE
261
......
275
276 /* NAND stuff */
277 #ifdef CONFIG_SYS_USE_NAND
278 #define CONFIG_CMD_NAND
279 #define CONFIG_CMD_NAND_TRIMFFS
280
281 #define CONFIG_NAND_MXS
282 #define CONFIG_SYS_MAX_NAND_DEVICE    1
283 #define CONFIG_SYS_NAND_BASE          0x40000000
284 #define CONFIG_SYS_NAND_5_ADDR_CYCLE
285 #define CONFIG_SYS_NAND_ONFI_DETECTION
286
287 /* DMA stuff, needed for GPMI/MXS NAND support */
288 #define CONFIG_APBH_DMA
289 #define CONFIG_APBH_DMA_BURST
290 #define CONFIG_APBH_DMA_BURST8
291 #endif
292
293 #define CONFIG_ENV_SIZE             SZ_8K
294 #if defined(CONFIG_ENV_IS_IN_MMC)
295 #define CONFIG_ENV_OFFSET           (12 * SZ_64K)
296 #elif defined(CONFIG_ENV_IS_IN_SPI_FLASH)
297 #define CONFIG_ENV_OFFSET           (768 * 1024)
298 #define CONFIG_ENV_SECT_SIZE        (64 * 1024)
299 #define CONFIG_ENV_SPI_BUS          CONFIG_SF_DEFAULT_BUS
300 #define CONFIG_ENV_SPI_CS           CONFIG_SF_DEFAULT_CS
301 #define CONFIG_ENV_SPI_MODE         CONFIG_SF_DEFAULT_MODE
302 #define CONFIG_ENV_SPI_MAX_HZ       CONFIG_SF_DEFAULT_SPEED
303 #elif defined(CONFIG_ENV_IS_IN_NAND)
304 #undef CONFIG_ENV_SIZE
305 #define CONFIG_ENV_OFFSET           (60 << 20)
306 #define CONFIG_ENV_SECT_SIZE        (128 << 10)
307 #define CONFIG_ENV_SIZE             CONFIG_ENV_SECT_SIZE
308 #endif
309
310
```

```
311 /* USB Configs */
312 #define CONFIG_CMD_USB
313 #ifdef CONFIG_CMD_USB
314 #define CONFIG_USB_EHCI
315 #define CONFIG_USB_EHCI_MX6
316 #define CONFIG_USB_STORAGE
317 #define CONFIG_EHCI_HCD_INIT_AFTER_RESET
318 #define CONFIG_USB_HOST_ETHER
319 #define CONFIG_USB_ETHER_ASIX
320 #define CONFIG_MXC_USB_PORTSC      (PORT_PTS_UTMI | PORT_PTS_PTW)
321 #define CONFIG_MXC_USB_FLAGS       0
322 #define CONFIG_USB_MAX_CONTROLLER_COUNT 2
323 #endif
324
325 #ifdef CONFIG_CMD_NET
326 #define CONFIG_CMD_PING
327 #define CONFIG_CMD_DHCP
328 #define CONFIG_CMD_MII
329 #define CONFIG_FEC_MXC
330 #define CONFIG_MII
331 #define CONFIG_FEC_ENET_DEV        1
332
333 #if (CONFIG_FEC_ENET_DEV == 0)
334 #define IMX_FEC_BASE               ENET_BASE_ADDR
335 #define CONFIG_FEC_MXC_PHYADDR     0x2
336 #define CONFIG_FEC_XCV_TYPE        RMII
337 #elif (CONFIG_FEC_ENET_DEV == 1)
338 #define IMX_FEC_BASE               ENET2_BASE_ADDR
339 #define CONFIG_FEC_MXC_PHYADDR     0x1
340 #define CONFIG_FEC_XCV_TYPE        RMII
341 #endif
342 #define CONFIG_ETHPRIME            "FEC"
343
344 #define CONFIG_PHYLIB
345 #define CONFIG_PHY_MICREL
346 #endif
347
348 #define CONFIG_IMX_THERMAL
349
350 #ifndef CONFIG_SPL_BUILD
351 #define CONFIG_VIDEO
352 #ifdef CONFIG_VIDEO
353 #define CONFIG_CFB_CONSOLE
```

```
354 #define CONFIG_VIDEO_MXS
355 #define CONFIG_VIDEO_LOGO
356 #define CONFIG_VIDEO_SW_CURSOR
357 #define CONFIG_VGA_AS_SINGLE_DEVICE
358 #define CONFIG_SYS_CONSOLE_IS_IN_ENV
359 #define CONFIG_SPLASH_SCREEN
360 #define CONFIG_SPLASH_SCREEN_ALIGN
361 #define CONFIG_CMD_BMP
362 #define CONFIG_BMP_16BPP
363 #define CONFIG_VIDEO_BMP_RLE8
364 #define CONFIG_VIDEO_BMP_LOGO
365 #define CONFIG_IMX_VIDEO_SKIP
366 #endif
367 #endif
368
369 #define CONFIG_IOMUX_LPSR
370
......
375 #endif
```

从示例代码 33.2.2.1 可以看出，mx6ull_alientek_emmc.h 文件中基本都是“CONFIG_”开头的宏定义，这也说明 mx6ull_alientek_emmc.h 文件的主要功能就是配置或者裁剪 uboot。如果需要某个功能的话就在里面添加这个功能对应的 CONFIG_XXX 宏即可，如果不需要某个功能的话就删除掉对应的宏即可。我们以示例代码 33.2.2.1 为例，详细的看一下 mx6ull_alientek_emmc.h 中这些宏都是什么功能。

第 14 行，添加了头文件 mx6_common.h，如果在 mx6ull_alientek_emmc.h 中没有发现有配置某个功能或命令，但是实际却存在的话，可以到 mx6_common.h 文件里面去找一下。

第 29~39 行，设置 DRAM 的大小，宏 PHYS_SDRAM_SIZE 就是板子上 DRAM 的大小，如果用的 NXP 官方的 9X9 EVK 开发板的话 DRAM 大小就为 256MB。否则的话默认为 512MB，正点原子的 I.MX6U-ALPHA 开发板用的是 512MB DDR3。

第 50 行，定义宏 CONFIG_DISPLAY_CPUINFO，uboot 启动的时候可以输出 CPU 信息。

第 51 行，定义宏 CONFIG_DISPLAY_BOARDINFO，uboot 启动的时候可以输出板子信息。

第 54 行，CONFIG_SYS_MALLOC_LEN 为 malloc 内存池大小，这里设置为 16MB。

第 56 行，定义宏 CONFIG_BOARD_EARLY_INIT_F，这样 board_init_f 函数就会调用 board_early_init_f 函数。

第 57 行，定义宏 CONFIG_BOARD_LATE_INIT，这样 board_init_r 函数就会调用 board_late_init 函数。

第 59、60 行，使能 I.MX6ULL 的串口功能，宏 CONFIG_MXC_UART_BASE 表示串口寄存器基地址，这里使用的串口 1，基地址为 UART1_BASE，UART1_BASE 定义在文件 arch/arm/include/asm/arch-mx6/imx-regs.h 中，imx-regs.h 是 I.MX6ULL 寄存器描述文件，根据 imx-regs.h 可得到 UART1_BASE 的值如下：

```
UART1_BASE= (ATZ1_BASE_ADDR + 0x20000)
           =AIPS1_ARB_BASE_ADDR + 0x20000
           =0x02000000 +   0x20000
```

```
=0X02020000
```

查阅 I.MX6ULL 参考手册，UART1 的寄存器基地址正是 0X02020000，如图 33.2.2.1 所示：

202_0000	UART Receiver Register (UART1_URXD)	32	R	0000_0000h	55.15.1/3615
202_0040	UART Transmitter Register (UART1_UTXD)	32	W	0000_0000h	55.15.2/3617
202_0080	UART Control Register 1 (UART1_UCR1)	32	R/W	0000_0000h	55.15.3/3618
202_0084	UART Control Register 2 (UART1_UCR2)	32	R/W	0000_0001h	55.15.4/3620

图 33.2.2.1 UART1 寄存器地址表

第 63、64 行，EMMC 接在 I.MX6ULL 的 USDHC2 上，宏 CONFIG_SYS_FSL_ESDHC_ADDR 为 EMMC 所使用接口的寄存器基地址，也就是 USDHC2 的基地址。

第 67~72 行，跟 NAND 相关的宏，因为 NAND 和 USDHC2 的引脚冲突，因此如果使用 NAND 的只能使用一个 USDHC 设备(SD 卡)。如果没有使用 NAND，那么就有两个 USDHC 设备(EMMC 和 SD 卡)，宏 CONFIG_SYS_FSL_USDHC_NUM 表示 USDHC 数量。EMMC 版本的核心版没有用到 NAND，所以 CONFIG_SYS_FSL_USDHC_NUM=2。

第 75~81，和 I2C 有关的宏定义，用于控制使能哪个 I2C，I2C 的速度为多少。

第 92~96 行，NAND 的分区设置，如果使用 NAND 的话，默认的 NAND 分区为："mtdparts=gpmi-nand:64m(boot),16m(kernel),16m(dtb),1m(misc),-(rootfs) "，分区结果如表 33.2.2.1 所示：

范围	大小	分区
0~63M	64M	boot(uboot)
64~79M	16M	kernel(linux 内核)
80~94M	16M	dtb(设备树)
95M	1M	misc(杂项)
96M – end	剩余的所有空间	rootfs(根文件系统)

表 33.2.2.1 NAND 分区设置

NAND 的分区是可以调整的，比如 boot 分区我们用不了 64M 这么大，因此可以将其改小，其他的分区一样的。

第 98~111 行，宏 CONFIG_MFG_ENV_SETTINGS 定义了一些环境变量，使用 MfgTool 烧写系统时候会用到这里面的环境变量。

第 113~202 行，通过条件编译来设置宏 CONFIG_EXTRA_ENV_SETTINGS，宏 CONFIG_EXTRA_ENV_SETTINGS 也是设置一些环境变量，此宏会设置 bootargs 这个环境变量，后面我们会详细分析这个宏定义。

第 204~217 行，设置宏 CONFIG_BOOTCOMMAND，此宏就是设置环境变量 bootcmd 的值。后面会详细的分析这个宏定义。

第 220~222 行，设置命令 memtest 相关宏定义，比如使能命令 memtest，设置 memtest 测试的内存起始地址和内存大小。

第 224 行，宏 CONFIG_SYS_LOAD_ADDR 表示 linux kernel 在 DRAM 中的加载地址，也就是 linux kernel 在 DRAM 中的存储首地址，CONFIG_LOADADDR=0X80800000。

第 225 行，宏 CONFIG_SYS_HZ 为系统时钟频率，这里为 1000Hz。

第 227 行，宏 CONFIG_STACKSIZE 为栈大小，这里为 128KB。

第 230 行，宏 CONFIG_NR_DRAM_BANKS 为 DRAM BANK 的数量，I.MX6ULL 只有一个 DRAM BANK，我们也只用到了一个 BANK，所以为 1。

第 231 行，宏 PHYS_SDRAM 为 I.MX6ULL 的 DRAM 控制器 MMDC0 所管辖的 DRAM 范围其实地址，也就是 0X80000000。

第 233 行，宏 CONFIG_SYS_SDRAM_BASE 为 DRAM 的其实地址。

第 234 行，宏 CONFIG_SYS_INIT_RAM_ADDR 为 I.MX6ULL 内部 IRAM 的起始地址(也就是 OCRAM 的起始地址)，为 0X00900000。

第 235 行，宏 CONFIG_SYS_INIT_RAM_SIZE 为 I.MX6ULL 内部 IRAM 的大小(OCRAM 的大小)，为 0X00040000=128KB。

第 237~240 行，宏 CONFIG_SYS_INIT_SP_OFFSET 和 CONFIG_SYS_INIT_SP_ADDR 与初始 SP 有关，第一个为初始 SP 偏移，第二个为初始 SP 地址。

第 256 行，宏 CONFIG_SYS_MMC_ENV_DEV 为默认的 MMC 设备，这里默认为 USDHC2，也就是 EMMC。

第 257 行，宏 CONFIG_SYS_MMC_ENV_PART 为模式分区，默认为第 0 个分区。

第 258 行，宏 CONFIG_MMCROOT 设置进入 linux 系统的根文件系统所在的分区，这里设置为"/dev/mmcblk1p2"，也就是 EMMC 设备的第 2 个分区。第 0 个分区保存 uboot，第 1 个分区保存 linux 镜像和设备树，第 2 个分区为 Linux 系统的根文件系统。

第 277~291 行，与 NAND 有关的宏定义，如果使用 NAND 的话。

第 293 行，宏 CONFIG_ENV_SIZE 为环境变量大小，默认为 8KB。

第 294~308 行，宏 CONFIG_ENV_OFFSET 为环境变量偏移地址，这里的偏移地址是相对于存储器的首地址。如果环境变量保存在 EMMC 中的话，环境变量偏移地址为 12*64KB。如果环境变量保存在 SPI FLASH 中的话，偏移地址为 768*1024。如果环境变量保存在 NAND 中的话，偏移地址为 60<<20(60MB)，并且重新设置环境变量的大小为 128KB。

第 312~323 行，与 USB 相关的宏定义。

第 325~342 行，与网络相关的宏定义，比如使能 dhcp、ping 等命令。第 331 行的宏 CONFIG_FEC_ENET_DEV 指定 uboot 所使用的网口，I.MX6ULL 有两个网口，为 0 的时候使用 ENET1，为 1 的时候使用 ENET2。宏 IMX_FEC_BASE 为 ENET 接口的寄存器首地址，宏 CONFIG_FEC_MXC_PHYADDR 为网口 PHY 芯片的地址。宏 CONFIG_FEC_XCV_TYPE 为 PHY 芯片所使用的接口类型，I.MX6U-ALPHA 开发板的两个 PHY 都使用的 RMII 接口。

第 344~END，剩下的都是一些配置宏，比如 CONFIG_VIDEO 宏用于开启 LCD，CONFIG_VIDEO_LOGO 使能 LOGO 显示，CONFIG_CMD_BMP 使能 BMP 图片显示指令。这样就可以在 uboot 中显示图片了，一般用于显示 logo。

关于 mx6ull_alientek_emmc.h 就讲解到这里，其中以 CONFIG_CMD 开头的宏都是用于使能相应命令的，其他的以 CONFIG 开头的宏都是完成一些配置功能的。以后会频繁的和 mx6ull_alientek_emmc.h 这个文件打交道。

33.2.3 添加开发板对应的板级文件夹

uboot 中每个板子都有一个对应的文件夹来存放板级文件，比如开发板上外设驱动文件等等。NXP 的 I.MX 系列芯片的所有板级文件夹都存放在 board/freescale 目录下，在这个目录下有个名为 mx6ullevk 的文件夹，这个文件夹就是 NXP 官方 I.MX6ULL EVK 开发板的板级文件夹。复制 mx6ullevk，将其重命名为 mx6ull_alientek_emmc，命令如下：

```
cd board/freescale/
cp mx6ullevk/ -r mx6ull_alientek_emmc
```

进入 mx6ull_alientek_emmc 目录中，将其中的 mx6ullevk.c 文件重命名为 mx6ull_alientek_emmc.c，命令如下：

```
cd mx6ull_alientek_emmc
mv mx6ullevk.c mx6ull_alientek_emmc.c
```

我们还需要对 mx6ull_alientek_emmc 目录下的文件做一些修改：

1、修改 mx6ull_alientek_emmc 目录下的 Makefile 文件

将 mx6ull_alientek_emmc 下的 Makefile 文件内容改为如下所示：

示例代码 33.2.3.1 Makefile 文件

```
1  # (C) Copyright 2015 Freescale Semiconductor, Inc.
2  #
3  # SPDX-License-Identifier:   GPL-2.0+
4  #
5
6  obj-y  := mx6ull_alientek_emmc.o
7
8  extra-$(CONFIG_USE_PLUGIN) :=  plugin.bin
9  $(obj)/plugin.bin: $(obj)/plugin.o
10     $(OBJCOPY) -O binary --gap-fill 0xff $< $@
```

重点是第 6 行的 obj-y，改为 mx6ull_alientek_emmc.o，这样才会编译 mx6ull_alientek_emmc.c 这个文件。

2、修改 mx6ull_alientek_emmc 目录下的 imximage.cfg 文件

将 imximage.cfg 中的下面一句：

```
PLUGIN board/freescale/mx6ullevk/plugin.bin 0x00907000
```

改为：

```
PLUGIN board/freescale/mx6ull_alientek_emmc /plugin.bin 0x00907000
```

3、修改 mx6ull_alientek_emmc 目录下的 Kconfig 文件

修改 Kconfig 文件，修改后的内容如下：

示例代码 33.2.3.2 Kconfig 文件

```
1  if TARGET_MX6ULL_ALIENTEK_EMMC
2
3  config SYS_BOARD
4   default "mx6ull_alientek_emmc"
5
6  config SYS_VENDOR
7   default "freescale"
8
9  config SYS_SOC
10  default "mx6"
11
12 config SYS_CONFIG_NAME
13  default "mx6ull_alientek_emmc"
14
15 endif
```

4、修改 mx6ull_alientek_emmc 目录下的 MAINTAINERS 文件

修改 MAINTAINERS 文件，修改后的内容如下：

```
1 MX6ULL_ALIENTEK_EMMC BOARD
2 M:    Peng Fan <peng.fan@nxp.com>
3 S:    Maintained
4 F:    board/freescale/mx6ull_alientek_emmc/
5 F:    include/configs/mx6ull_alientek_emmc.h
```

33.2.4 修改 U-Boot 图形界面配置文件

uboot 是支持图形界面配置，关于 uboot 的图形界面配置下一章会详细的讲解。修改文件 arch/arm/cpu/armv7/mx6/Kconfig(如果用的 I.MX6UL 的话，应该修改 arch/arm/Kconfig 这个文件)，在 207 行加入如下内容：

示例代码 33.2.4.1 Kconfig 文件

```
1 config TARGET_MX6ULL_ALIENTEK_EMMC
2       bool "Support mx6ull_alientek_emmc"
3       select MX6ULL
4       select DM
5       select DM_THERMAL
```

在最后一行的 endif 的前一行添加如下内容：

示例代码 33.2.4.2 Kconfig 文件

```
1 source "board/freescale/mx6ull_alientek_emmc/Kconfig"
```

添加完成以后的 Kconfig 文件如图 33.2.4.1 所示：

```
201 config TARGET_MX6ULL_9X9_EVK
202     bool "Support mx6ull_9x9_evk"
203     select MX6ULL
204     select DM
205     select DM_THERMAL
206
207 config TARGET_MX6ULL_ALIENTEK_EMMC
208     bool "Support mx6ull_alientek_emmc"
209     select MX6ULL
210     select DM
211     select DM_THERMAL
212
284 source "board/warp/Kconfig"
285 source "board/freescale/mx6dqscm/Kconfig"
286 source "board/freescale/mx6sxscm/Kconfig"
287
288 source "board/freescale/mx6ul_alientek_emmc/Kconfig"
289
290 endif
291
```

图 33.2.4.1 修改后的 Kconfig 文件

到此为止，I.MX6U-ALPHA 开发板就已经添加到 uboot 中了，接下来就是编译这个新添加的开发板。

33.2.5 使用新添加的板子配置编译 uboot

在 uboot 根目录下新建一个名为 mx6ull_alientek_emmc.sh 的 shell 脚本，在这个 shell 脚本里面输入如下内容：

示例代码 33.2.5.1 mx6ull_alientek_emmc.sh 脚本文件

```
1 #!/bin/bash
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
mx6ull_alientek__emmc_defconfig
4 make V=1  ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j16
```

第 3 行我们使用的默认配置文件就是 33.2.1 节中新建的 mx6ull_alientek_emmc_defconfig 这个配置文件。给予 mx6ll_alientek_emmc.sh 可执行权限，然后运行脚本来完成编译，命令如下：

```
chmod 777 mx6ull_alientek_emmc.sh        //给予可执行权限，一次即可
./mx6ull_alientek_emmc.sh                //运行脚本编译 uboot
```

等待编译完成，编译完成以后输入如下命令，查看一下 33.2.2 小节中添加的 mx6ull_alientek_emmc.h 这个头文件有没有被引用。

```
grep -nR "mx6ull_alientek_emmc.h"
```

如果有很多文件都引用了 mx6ull_alientek_emmc.h 这个头文件，那就说明新板子添加成功，如图 33.2.5.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/uboot/temp/uboot-imx-rel_imx_4.1.15_2.1.0_ga/arch$ grep -nR "mx6ull_alientek_emmc.h"
arm/lib/.reset.o.cmd:133:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.cache-cp15.o.cmd:139:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.relocate.o.cmd:44:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.crt0.o.cmd:47:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.stack.o.cmd:137:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.vectors.o.cmd:42:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.eabi_compat.o.cmd:133:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.bootm-fdt.o.cmd:135:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.asm-offsets.s.cmd:138:  include/configs/mx6ull_alientek_emmc.h \
arm/lib/.interrupts.o.cmd:135:  include/configs/mx6ull_alientek_emmc.h \
```

图 33.2.5.1 查找结果

编译完成以后就使用 imxdownload 将新编译出来的 u-boot.bin 烧写到 SD 卡中测试，SecureCRT 输出结果如图 33.2.5.2 所示：

```
serial-com12

U-Boot 2016.03 (May 12 2019 - 22:17:47 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 48C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
unsupported panel TFT7016
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
Net:   Board Net Initialization Failed
No ethernet found.
Normal Boot
Hit any key to stop autoboot:  0
=>
```

图 33.2.5.1 uboot 启动过程

从图 33.2.5.1 可以看出，此时的 Board 还是"MX6ULL 14x14 EVK"，因为我们参考的 NXP 官方的 I.MX6ULL 开发板来添加自己的开发板。如果接了 LCD 屏幕的话会发现 LCD 屏幕并没有显示 NXP 的 logo，而且从图 33.2.5.1 可以看出此时的网络同样也没识别出来。前面已经说

了，默认 uboot 中的 LCD 驱动和网络驱动在正点原子的 I.MX6U-ALPHA 开发板上是有问题的，需要修改。

33.2.6 LCD 驱动修改

一般 uboot 中修改驱动基本都是在 xxx.h 和 xxx.c 这两个文件中进行的，xxx 为板子名称，比如 mx6ull_alientek_emmc.h 和 mx6ull_alientek_emmc.c 这两个文件。

一般修改 LCD 驱动重点注意以下几点：

①、LCD 所使用的 GPIO，查看 uboot 中 LCD 的 IO 配置是否正确。

②、LCD 背光引脚 GPIO 的配置。

③、LCD 配置参数是否正确。

正点原子的 I.MX6U-ALPHA 开发板 LCD 原理图和 NXP 官方 I.MX6ULL 开发板一致，也就是 LCD 的 IO 和背光 IO 都一样的，所以 IO 部分就不用修改了。需要修改的之后 LCD 参数，打开文件 mx6ull_alientek_emmc.c，找到如下所示内容：

示例代码 33.2.6.1 LCD 驱动参数

```
1  struct display_info_t const displays[] = {{
2      .bus = MX6UL_LCDIF1_BASE_ADDR,
3      .addr = 0,
4      .pixfmt = 24,
5      .detect = NULL,
6      .enable = do_enable_parallel_lcd,
7      .mode   = {
8          .name            = "TFT43AB",
9          .xres            = 480,
10         .yres            = 272,
11         .pixclock        = 108695,
12         .left_margin     = 8,
13         .right_margin    = 4,
14         .upper_margin    = 2,
15         .lower_margin    = 4,
16         .hsync_len       = 41,
17         .vsync_len       = 10,
18         .sync            = 0,
19         .vmode           = FB_VMODE_NONINTERLACED
20 } } };
```

示例代码 33.2.6.1 中定义了一个变量 displays，类型为 display_info_t，这个结构体是 LCD 信息结构体，其中包括了 LCD 的分辨率，像素格式，LCD 的各个参数等。display_info_t 定义在文件 arch/arm/include/asm/imx-common/video.h 中，定义如下：

示例代码 33.2.6.2 display_info 结构体

```
1 struct display_info_t {
2     int bus;
3     int addr;
4     int pixfmt;
5     int (*detect)(struct display_info_t const *dev);
```

```
6       void    (*enable)(struct display_info_t const *dev);
7       struct  fb_videomode mode;
8  };
```

pixfmt 是像素格式，也就是一个像素点是多少位，如果是 RGB565 的话就是 16 位，如果是 888 的话就是 24 位，一般使用 RGB888。结构体 display_info_t 还有个 mode 成员变量，此成员变量也是个结构体，为 fb_videomode，定义在文件 include/linux/fb.h 中，定义如下：

示例代码 33.2.6.3 fb_videomode 结构体

```
1  struct fb_videomode {
2       const char *name;     /* optional */
3       u32 refresh;          /* optional */
4       u32 xres;
5       u32 yres;
6       u32 pixclock;
7       u32 left_margin;
8       u32 right_margin;
9       u32 upper_margin;
10      u32 lower_margin;
11      u32 hsync_len;
12      u32 vsync_len;
13      u32 sync;
14      u32 vmode;
15      u32 flag;
16 };
```

结构体 fb_videomode 里面的成员变量为 LCD 的参数，这些成员变量函数如下：

name：LCD 名字，要和环境变量中的 panel 相等。

xres、**yres**：LCD X 轴和 Y 轴像素数量。

pixclock：像素时钟，每个像素时钟周期的长度，单位为皮秒。

left_margin：HBP，水平同步后肩。

right_margin：HFP，水平同步前肩。

upper_margin：VBP，垂直同步后肩。

lower_margin：VFP，垂直同步前肩。

hsync_len：HSPW，行同步脉宽。

vsync_len：VSPW，垂直同步脉宽。

vmode：大多数使用 FB_VMODE_NONINTERLACED，也就是不使用隔行扫描。

可以看出，这些参数和我们第二十四章讲解 RGB LCD 的时候参数基本一样，唯一不同的像素时钟 pixclock 的含义不同，以正点原子的 7 寸 1024*600 分辨率的屏幕(ATK7016)为例，屏幕要求的像素时钟为 51.2MHz，因此：

```
pixclock=(1/51200000)*10^12=19531
```

在根据其他的屏幕参数，可以得出 ATK7016 屏幕的配置参数如下：

示例代码 33.2.6.4 ATK7016 屏幕配置参数

```
1  struct display_info_t const displays[] = {{
2       .bus = MX6UL_LCDIF1_BASE_ADDR,
3       .addr = 0,
```

```
4       .pixfmt = 24,
5       .detect = NULL,
6       .enable = do_enable_parallel_lcd,
7       .mode   = {
8           .name           = "TFT7016",
9           .xres           = 1024,
10          .yres           = 600,
11          .pixclock       = 19531,
12          .left_margin    = 140,        //HBPD
13          .right_margin   = 160,        //HFPD
14          .upper_margin   = 20,         //VBPD
15          .lower_margin   = 12,         //VFBD
16          .hsync_len      = 20,         //HSPW
17          .vsync_len      = 3,          //VSPW
18          .sync           = 0,
19          .vmode          = FB_VMODE_NONINTERLACED
20 } } };
```

使用示例代码 33.2.6.4 中的屏幕参数替换掉 mx6ull_alientek_emmc.c 中 uboot 默认的屏幕参数。

打开 mx6ull_alientek_emmc.h，找到所有如下语句：

```
panel=TFT43AB
```

将其改为：

```
panel=TFT7016
```

也就是设置 panel 为 TFT7016，panel 的值要与示例代码 33.2.6.4 中的.name 成员变量的值一致。修改完成以后重新编译一遍 uboot 并烧写到 SD 中启动。

重启以后 LCD 驱动一般就会工作正常了，LCD 上回显示 NXP 的 logo。但是有可能会遇到 LCD 并没有工作，还是黑屏，这是什么原因呢？在 uboot 命令模式输入 "printf" 来查看环境变量 panel 的值，会发现 panel 的值要是 TFT43AB(或其他的，反正不是 TFT7016)，如图 33.2.6.1 所示：

```
panel=TFT43AB
script=boot.scr
serverip=192.168.1.250

Environment size: 2639/8188 bytes
=>
```

图 33.2.6.1 panel 的值

这是因为之前有将环境变量保存到 EMMC 中，uboot 启动以后会先从 EMMC 中读取环境变量，如果 EMMC 中没有环境变量的话才会使用 mx6ull_alientek_emmc.h 中的默认环境变量。如果 EMMC 中的环境变量 panel 不等于 TFT7016，那么 LCD 显示肯定不正常，我们只需要在 uboot 中修改 panel 的值为 TFT7016 即可，在 uboot 的命令模式下输入如下命令：

```
setenv panel TFT7016
saveenv
```

上述命令修改环境变量 panel 为 TFT7016，然后保存，重启 uboot，此时 LCD 驱动就工作正常了。如果 LCD 还是没有正常工作的，那就要检查自己哪里有没有改错，或者还有哪里没有修改。

33.2.7 网络驱动修改

1、I.MX6U-ALPHA 开发板网络简介

I.MX6UL/ULL 内部有个以太网 MAC 外设，也就是 ENET，需要外接一个 PHY 芯片来实现网络通信功能，也就是内部 MAC+外部 PHY 芯片的方案。大家可能听过 DM9000 这个网络芯片，在一些没有内部 MAC 的 CPU 中，比如三星的 2440，4412 等，就会采用 DM9000 来实现联网功能。DM9000 提供了一个类似 SRAM 的访问接口，主控 CPU 通过这个接口即可与 DM9000 进行通信，DM9000 就是一个 MAC+PHY 芯片。这个方案就相当于外部 MAC+外部 PHY，那么 I.MX6U 这样的内部 MAC+PHY 芯片与 DM9000 方案比有什么优势吗？那优势大了去了！首先就是通信效率和速度，一般 CPU 内部的 MAC 是带有一个专用 DMA 的，专门用于处理网络数据包，采用 SRAM 来读写 DM9000 的速度是压根就没法和内部 MAC+外部 PHY 芯片的速度比。采用外部 DM9000 完全是无奈之举，谁让 2440，4412 这些芯片内部没有以太网外设呢，现在又想用有线网络，没有办法只能找个 DM9000 的方案。从这里也可以看出，三星的 2440、4412 这些芯片设计之初就不是给工业产品用的，他们是给消费类电子使用的，比如手机、平板等，手机或平板要上网，可以通过 WIFI 或者 4G，我是没有见过哪个手机或者平板上网是要接根网线的。正点原子的 I.MX6U-ALPHA 开发板也可以通过 WIFI 或者 4G 上网，这个是后话了。

I.MX6UL/ULL 有两个网络接口 ENET1 和 ENET2，正点原子的 I.MX6U-ALPHA 开发板提供了这两个网络接口，其中 ENET1 和 ENET2 都使用 LAN8720A 作为 PHY 芯片。NXP 官方的 I.MX6ULL EVK 开发板使用 KSZ8081 这颗 PHY 芯片，LAN8720A 相比 KSZ8081 具有体积小、外围器件少、价格便宜等优点。直接使用 KSZ8081 固然可以，但是我们在实际的产品中不一定会使用 KSZ8081，有时候为了降低成本会选择其他的 PHY 芯片，这个时候就有个问题：换了 PHY 芯片以后网络驱动怎么办？为此，正点原子的 I.MX6U-ALPHA 开发板将 ENET1 和 ENET2 的 PHY 换为了 LAN8720A，这样就可以给大家讲解更换 PHY 芯片以后如何调整网络驱动，使网络工作正常。

I.MX6U-ALPHA 开发板 ENET1 的网络原理图如图 33.2.7.1 所示：

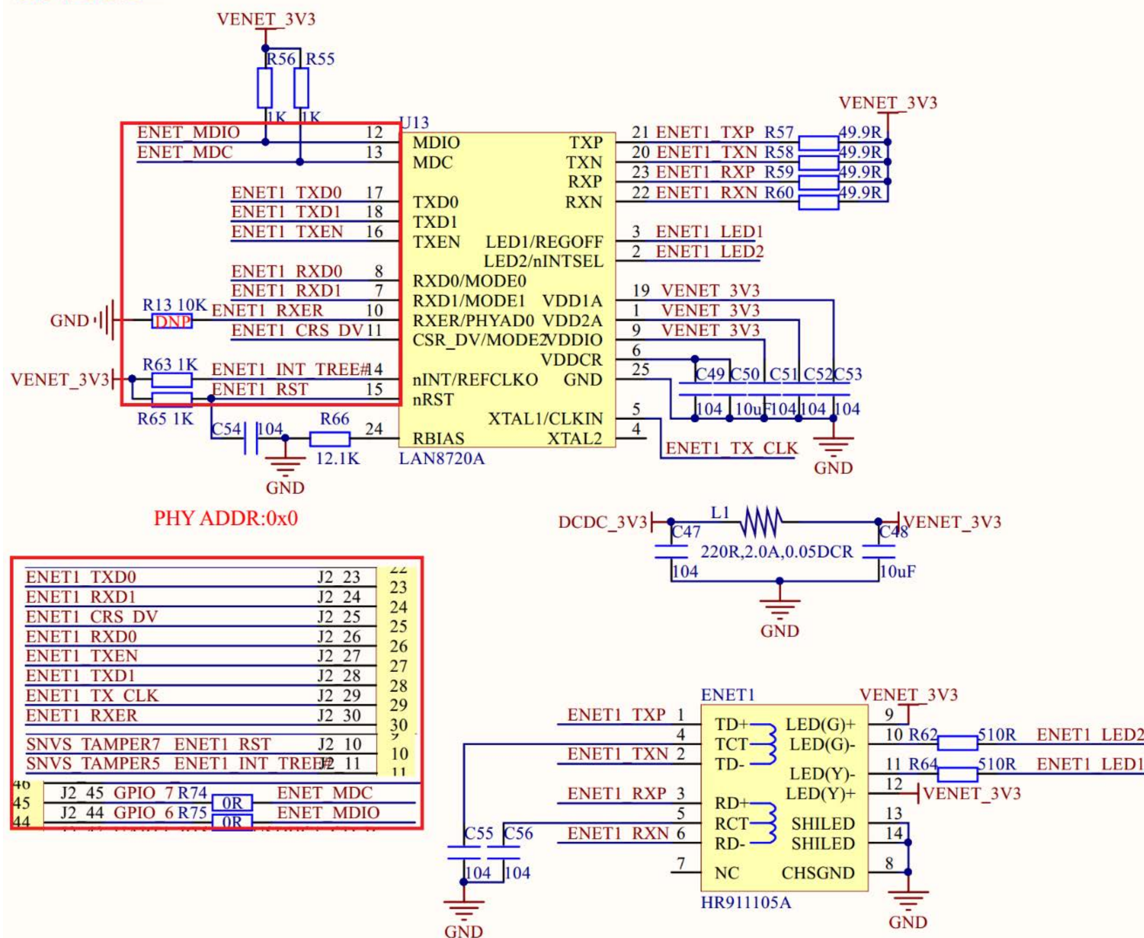


图 33.2.7.1 ENET1 原理图

ENET1 的网络 PHY 芯片为 LAN8720A，通过 RMII 接口与 I.MX6ULL 相连，正点原子 I.MX6U-ALPHA 开发板的 ENET1 引脚与 NXP 官方的 I.MX6ULL EVK 开发板基本一样，唯独复位引脚不同。从图 33.2.7.1 可以看出，正点原子 I.MX6U-ALPHA 开发板的 ENET1 复位引脚 ENET1_RST 接到了 I.M6ULL 的 SNVS_TAMPER7 这个引脚上。

LAN8720A 内部是有寄存器的，I.MX6ULL 会读取 LAN8720 内部寄存器来判断当前的物理链接状态、连接速度(10M 还是 100M)和双工状态(半双工还是全双工)。I.MX6ULL 通过 MDIO 接口来读取 PHY 芯片的内部寄存器，MDIO 接口有两个引脚，ENET_MDC 和 ENET_MDIO，ENET_MDC 提供时钟，ENET_MDIO 进行数据传输。一个 MIDO 接口可以管理 32 个 PHY 芯片，同一个 MDIO 接口下的这些 PHY 使用不同的器件地址来做区分，MIDO 接口通过不同的器件地址即可访问到相应的 PHY 芯片。I.MX6U-ALPHA 开发板 ENET1 上连接的 LAN8720A 器件地址为 0X0，所示我们要修改 ENET1 网络驱动的话重点就三点：

①、ENET1 复位引脚初始化。

②、LAN8720A 的器件 ID。

③、LAN8720 驱动

再来看一下 ENET2 的原理图，如图 33.2.7.2 所示：

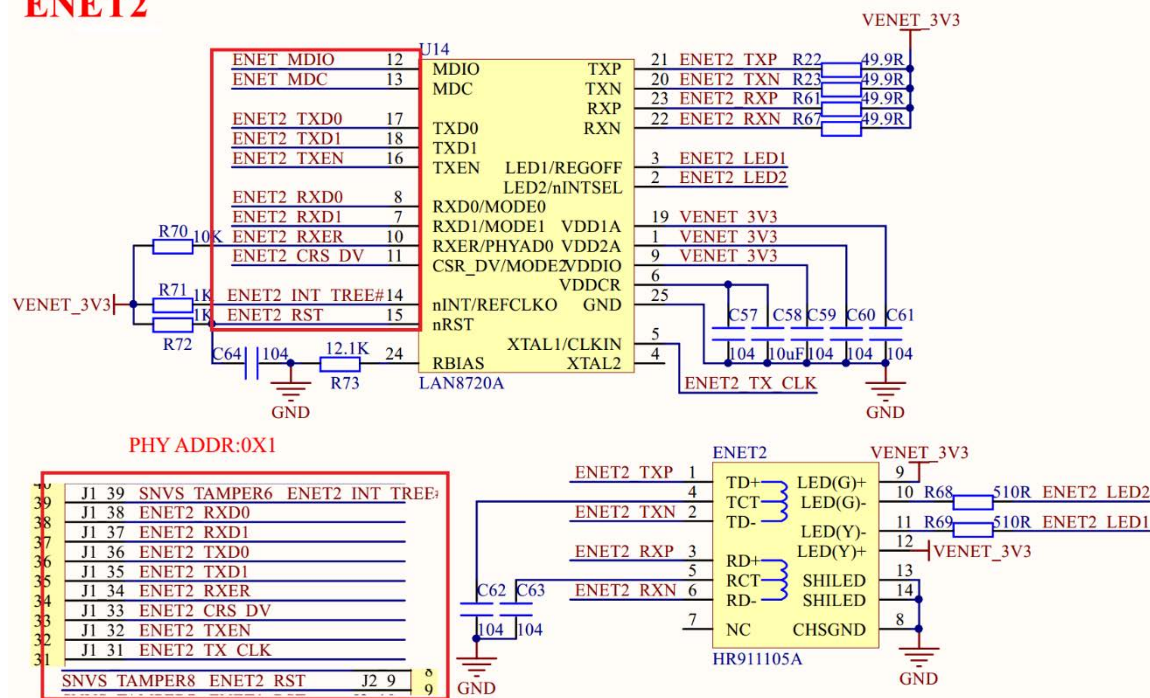


图 33.2.7.2 ENET2 原理图

关于 ENET2 网络驱动的修改也注意一下三点：

①、ENET2 的复位引脚，从图 33.2.7.2 可以看出，ENET2 的复位引脚 ENET2_RST 接到了 I.MX6ULL 的 SNVS_TAMPER8 上。

②、ENET2 所使用的 PHY 芯片器件地址，从图 33.2.7.2 可以看出，PHY 器件地址为 0X1。

③、LAN8720 驱动，ENET1 和 ENET2 都使用的 LAN8720，所以驱动肯定是一样的。

2、网络 PHY 地址修改

首先修改 uboot 中的 ENET1 和 ENET2 的 PHY 地址和驱动，打开 mx6ull_alientek_emmc.h 这个文件，找到如下代码：

示例代码 33.2.7.1 网络默认 ID 配置参数

```
325 #ifdef CONFIG_CMD_NET
326 #define CONFIG_CMD_PING
327 #define CONFIG_CMD_DHCP
328 #define CONFIG_CMD_MII
329 #define CONFIG_FEC_MXC
330 #define CONFIG_MII
331 #define CONFIG_FEC_ENET_DEV          1
332
333 #if (CONFIG_FEC_ENET_DEV == 0)
334 #define IMX_FEC_BASE                 ENET_BASE_ADDR
335 #define CONFIG_FEC_MXC_PHYADDR       0x2
336 #define CONFIG_FEC_XCV_TYPE          RMII
337 #elif (CONFIG_FEC_ENET_DEV == 1)
338 #define IMX_FEC_BASE                 ENET2_BASE_ADDR
339 #define CONFIG_FEC_MXC_PHYADDR       0x1
```

```
340 #define CONFIG_FEC_XCV_TYPE          RMII
341 #endif
342 #define CONFIG_ETHPRIME              "FEC"
343
344 #define CONFIG_PHYLIB
345 #define CONFIG_PHY_MICREL
346 #endif
```

第 331 行的宏 CONFIG_FEC_ENET_DEV 用于选择使用哪个网口，默认为 1，也就是选择 ENET2。第 335 行为 ENET1 的 PHY 地址，默认是 0X2，第 339 行为 ENET2 的 PHY 地址，默认为 0x1。根据前面的分析可知，正点原子的 I.MX6U-ALPHA 开发板 ENET1 的 PHY 地址为 0X0，ENET2 的 PHY 地址为 0X1，所以需要将第 335 行的宏 CONFIG_FEC_MXC_PHYADDR 改为 0x0。

第 345 行定了一个宏 CONFIG_PHY_MICREL，此宏用于使能 uboot 中 Micrel 公司的 PHY 驱动，KSZ8081 这颗 PHY 芯片就是 Micrel 公司生产的，不过 Micrel 已经被 Microchip 收购了。如果要使用 LAN8720A，那么就得将 CONFIG_PHY_MICREL 改为 CONFIG_PHY_SMSC，也就是使能 uboot 中的 SMSC 公司中的 PHY 驱动，因为 LAN8720A 就是 SMSC 公司生产的。所以示例代码 33.2.7.1 有三处要修改：

①、修改 ENET1 网络 PHY 的地址。

②、修改 ENET2 网络 PHY 的地址。

③、使能 SMSC 公司的 PHY 驱动。

修改后的网络 PHY 地址参数如下所示：

示例代码 33.2.7.2 网络 PHY 地址配置参数

```
325 #ifdef CONFIG_CMD_NET
326 #define CONFIG_CMD_PING
327 #define CONFIG_CMD_DHCP
328 #define CONFIG_CMD_MII
329 #define CONFIG_FEC_MXC
330 #define CONFIG_MII
331 #define CONFIG_FEC_ENET_DEV          1
332
333 #if (CONFIG_FEC_ENET_DEV == 0)
334 #define IMX_FEC_BASE                 ENET_BASE_ADDR
335 #define CONFIG_FEC_MXC_PHYADDR       0x0
336 #define CONFIG_FEC_XCV_TYPE          RMII
337 #elif (CONFIG_FEC_ENET_DEV == 1)
338 #define IMX_FEC_BASE                 ENET2_BASE_ADDR
339 #define CONFIG_FEC_MXC_PHYADDR       0x1
340 #define CONFIG_FEC_XCV_TYPE          RMII
341 #endif
342 #define CONFIG_ETHPRIME              "FEC"
343
344 #define CONFIG_PHYLIB
345 #define CONFIG_PHY_SMSC
```

```
346 #endif
```

3、删除 uboot 中 74LV595 的驱动代码

uboot 中网络 PHY 芯片地址修改完成以后就是网络复位引脚的驱动修改了，打开 mx6ull_alientek_emmc.c，找到如下代码：

示例代码 33.2.7.3 74LV595 引脚

```
#define IOX_SDI   IMX_GPIO_NR(5, 10)
#define IOX_STCP  IMX_GPIO_NR(5, 7)
#define IOX_SHCP  IMX_GPIO_NR(5, 11)
#define IOX_OE    IMX_GPIO_NR(5, 8)
```

示例代码 33.2.7.3 中以 IOX 开头的宏定义是 74LV595 的相关 GPIO，因为 NXP 官方 I.MX6ULL EVK 开发板使用 74LV595 来扩展 IO，两个网络的复位引脚就是由 74LV595 来控制的。正点原子的 I.MX6U-ALPHA 开发板并没有使用 74LV595，因此我们将示例代码 33.2.6.6 中的代码删除掉，替换为如下所示代码：

示例代码 33.2.7.4 修改后的网络引脚

```
#define ENET1_RESET IMX_GPIO_NR(5, 7)
#define ENET2_RESET IMX_GPIO_NR(5, 8)
```

ENET1 的复位引脚连接到 SNVS_TAMPER7 上，对应 GPIO5_IO07，ENET2 的复位引脚连接到 SNVS_TAMPER8 上，对应 GPIO5_IO08。

继续在 mx6ull_alientek_emmc.c 中找到如下代码：

示例代码 33.2.7.5 74LV595 引脚配置

```
static iomux_v3_cfg_t const iox_pads[] = {
    /* IOX_SDI */
    MX6_PAD_BOOT_MODE0__GPIO5_IO10 | MUX_PAD_CTRL(NO_PAD_CTRL),
    /* IOX_SHCP */
    MX6_PAD_BOOT_MODE1__GPIO5_IO11 | MUX_PAD_CTRL(NO_PAD_CTRL),
    /* IOX_STCP */
    MX6_PAD_SNVS_TAMPER7__GPIO5_IO07 | MUX_PAD_CTRL(NO_PAD_CTRL),
    /* IOX_nOE */
    MX6_PAD_SNVS_TAMPER8__GPIO5_IO08 | MUX_PAD_CTRL(NO_PAD_CTRL),
};
```

同理，示例代码 33.2.7.5 是 74LV595 的 IO 配置参数结构体，将其删除掉。继续在 mx6ull_alientek_emmc.c 中找到函数 iox74lv_init，如下所示：

示例代码 33.2.7.6 74LV595 初始化函数

```
static void iox74lv_init(void)
{
    int i;

    gpio_direction_output(IOX_OE, 0);

    for (i = 7; i >= 0; i--) {
        gpio_direction_output(IOX_SHCP, 0);
        gpio_direction_output(IOX_SDI, seq[qn_output[i]][0]);
        udelay(500);
```

```
        gpio_direction_output(IOX_SHCP, 1);
        udelay(500);
    }

    ......
    /*
     * shift register will be output to pins
     */
    gpio_direction_output(IOX_STCP, 1);
};

void iox74lv_set(int index)
{
    int i;

    for (i = 7; i >= 0; i--) {
        gpio_direction_output(IOX_SHCP, 0);

        if (i == index)
            gpio_direction_output(IOX_SDI, seq[qn_output[i]][0]);
        else
            gpio_direction_output(IOX_SDI, seq[qn_output[i]][1]);
        udelay(500);
        gpio_direction_output(IOX_SHCP, 1);
        udelay(500);
    }
    ......
    /*
      * shift register will be output to pins
      */
    gpio_direction_output(IOX_STCP, 1);
};
```

iox74lv_init 函数是 74LV595 的初始化函数，iox74lv_set 函数用于控制 74LV595 的 IO 输出电平，将这两个函数全部删除掉！

在 mx6ull_alientek_emmc.c 中找到 board_init 函数，此函数是板子初始化函数，会被 board_init_r 调用，board_init 函数内容如下：

示例代码 33.2.7.7 board_init 函数

```
int board_init(void)
{
    ......
    imx_iomux_v3_setup_multiple_pads(iox_pads, ARRAY_SIZE(iox_pads));
    iox74lv_init();
    ......
```

```
    return 0;
}
```

board_init 会调用 imx_iomux_v3_setup_multiple_pads 和 iox74lv_init 这两个函数来初始化 74lv595 的 GPIO，将这两行删除掉。至此，mx6ull_alientek_emmc.c 中关于 74LV595 芯片的驱动代码都删除掉了，接下来就是添加 I.MX6U-ALPHA 开发板两个网络复位引脚了。

4、添加 I.MX6U-ALPHA 开发板网络复位引脚驱动

在 mx6ull_alientek_emmc.c 中找到如下所示代码：

示例代码 33.2.7.8 默认网络 IO 结构体数组

```
640 static iomux_v3_cfg_t const fec1_pads[] = {
641     MX6_PAD_GPIO1_IO06__ENET1_MDIO | MUX_PAD_CTRL(MDIO_PAD_CTRL),
642     MX6_PAD_GPIO1_IO07__ENET1_MDC | MUX_PAD_CTRL(ENET_PAD_CTRL),
......
649     MX6_PAD_ENET1_RX_ER__ENET1_RX_ER | MUX_PAD_CTRL(ENET_PAD_CTRL),
650     MX6_PAD_ENET1_RX_EN__ENET1_RX_EN | MUX_PAD_CTRL(ENET_PAD_CTRL),
651 };
652
653 static iomux_v3_cfg_t const fec2_pads[] = {
654     MX6_PAD_GPIO1_IO06__ENET2_MDIO | MUX_PAD_CTRL(MDIO_PAD_CTRL),
655     MX6_PAD_GPIO1_IO07__ENET2_MDC | MUX_PAD_CTRL(ENET_PAD_CTRL),
......
664     MX6_PAD_ENET2_RX_EN__ENET2_RX_EN | MUX_PAD_CTRL(ENET_PAD_CTRL),
665     MX6_PAD_ENET2_RX_ER__ENET2_RX_ER | MUX_PAD_CTRL(ENET_PAD_CTRL),
666 };
```

结构体数组 fec1_pads 和 fec2_pads 是 ENET1 和 ENET2 这两个网口的 IO 配置参数，在这两个数组中添加两个网口的复位 IO 配置参数，完成以后如下所示：

示例代码 33.2.7.9 添加网络复位 IO 后的结构体数组

```
640 static iomux_v3_cfg_t const fec1_pads[] = {
641     MX6_PAD_GPIO1_IO06__ENET1_MDIO | MUX_PAD_CTRL(MDIO_PAD_CTRL),
642     MX6_PAD_GPIO1_IO07__ENET1_MDC | MUX_PAD_CTRL(ENET_PAD_CTRL),
......
649     MX6_PAD_ENET1_RX_ER__ENET1_RX_ER | MUX_PAD_CTRL(ENET_PAD_CTRL),
650     MX6_PAD_ENET1_RX_EN__ENET1_RX_EN | MUX_PAD_CTRL(ENET_PAD_CTRL),
651    MX6_PAD_SNVS_TAMPER7__GPIO5_IO07 | MUX_PAD_CTRL(NO_PAD_CTRL),
652 };
653
654 static iomux_v3_cfg_t const fec2_pads[] = {
655     MX6_PAD_GPIO1_IO06__ENET2_MDIO | MUX_PAD_CTRL(MDIO_PAD_CTRL),
656     MX6_PAD_GPIO1_IO07__ENET2_MDC | MUX_PAD_CTRL(ENET_PAD_CTRL),
......
665     MX6_PAD_ENET2_RX_EN__ENET2_RX_EN | MUX_PAD_CTRL(ENET_PAD_CTRL),
666     MX6_PAD_ENET2_RX_ER__ENET2_RX_ER | MUX_PAD_CTRL(ENET_PAD_CTRL),
667     MX6_PAD_SNVS_TAMPER8__GPIO5_IO08 | MUX_PAD_CTRL(NO_PAD_CTRL),
668 };
```

示例代码 33.2.7.9 中，第 651 行和 667 行分别是 ENET1 和 ENET2 的复位 IO 配置参数。继续在文件 mx6ull_alientek_emmc.c 中找到函数 setup_iomux_fec，此函数默认代码如下：

示例代码 33.2.7.10 setup_iomux_fec 函数默认代码
```
668 static void setup_iomux_fec(int fec_id)
669 {
670     if (fec_id == 0)
671         imx_iomux_v3_setup_multiple_pads(fec1_pads,
672                               ARRAY_SIZE(fec1_pads));
673     else
674         imx_iomux_v3_setup_multiple_pads(fec2_pads,
675                               ARRAY_SIZE(fec2_pads));
676 }
```

函数 setup_iomux_fec 就是根据 fec1_pads 和 fec2_pads 这两个网络 IO 配置数组来初始化 I.MX6ULL 的网络 IO。我们需要在其中添加网络复位 IO 的初始化代码，并且复位一下 PHY 芯片，修改后的 setup_iomux_fec 函数如下：

示例代码 33.2.7.11 修改后的 setup_iomux_fec 函数
```
668 static void setup_iomux_fec(int fec_id)
669 {
670     if (fec_id == 0)
671     {
672 
673         imx_iomux_v3_setup_multiple_pads(fec1_pads,
674                               ARRAY_SIZE(fec1_pads));
675 
676         gpio_direction_output(ENET1_RESET, 1);
677         gpio_set_value(ENET1_RESET, 0);
678         mdelay(20);
679         gpio_set_value(ENET1_RESET, 1);
680     }
681     else
682     {
683         imx_iomux_v3_setup_multiple_pads(fec2_pads,
684                               ARRAY_SIZE(fec2_pads));
685         gpio_direction_output(ENET2_RESET, 1);
686         gpio_set_value(ENET2_RESET, 0);
687         mdelay(20);
688         gpio_set_value(ENET2_RESET, 1);
689     }
690 }
```

示例代码 33.2.7.11 中第 676 行~679 行和第 685 行~688 行分别对应 ENET1 和 ENET2 的复位 IO 初始化，将这两个 IO 设置为输出并且硬件复位一下 LAN8720A，这个硬件复位很重要！否则可能导致 uboot 无法识别 LAN8720A。

5、修改 drivers/net/phy/phy.c 文件中的函数 genphy_update_link

大功基本上告成，还差最后一步，uboot 中的 LAN8720A 驱动有点问题，打开文件 drivers/net/phy/phy.c，找到函数 genphy_update_link，这是个通用 PHY 驱动函数，此函数用于更新 PHY 的连接状态和速度。使用 LAN8720A 的时候需要在此函数中添加一些代码，修改后的函数 genphy_update_link 如下所示：

示例代码 33.2.7.12 修改后的 genphy_update_link 函数

```
221 int genphy_update_link(struct phy_device *phydev)
222 {
223     unsigned int mii_reg;
224 
225 #ifdef CONFIG_PHY_SMSC
226     static int lan8720_flag = 0;
227     int bmcr_reg = 0;
228     if (lan8720_flag == 0) {
229         bmcr_reg = phy_read(phydev, MDIO_DEVAD_NONE, MII_BMCR);
230         phy_write(phydev, MDIO_DEVAD_NONE, MII_BMCR, BMCR_RESET);
231         while(phy_read(phydev, MDIO_DEVAD_NONE, MII_BMCR) & 0X8000) {
232             udelay(100);
233         }
234         phy_write(phydev, MDIO_DEVAD_NONE, MII_BMCR, bmcr_reg);
235         lan8720_flag = 1;
236     }
237 #endif
238 
239     /*
240      * Wait if the link is up, and autonegotiation is in progress
241      * (ie - we're capable and it's not done)
242      */
243     mii_reg = phy_read(phydev, MDIO_DEVAD_NONE, MII_BMSR);
......
291 
292     return 0;
293 }
```

225 行~237 行就是新添加的代码，为条件编译代码段，只有使用 SMSC 公司的 PHY 这段代码才会执行(目前只测试了 LAN8720A，SMSC 公司其他的芯片还未测试)。第 229 行读取 LAN8720A 的 BMCR 寄存器(寄存器地址为 0)，此寄存器为 LAN8720A 的配置寄存器，这里先读取此寄存器的默认值并保存起来。230 行向寄存器 BMCR 寄存器写入 BMCR_RESET(值为 0X8000)，因为 BMCR 的 bit15 是软件复位控制位，因此 230 行就是软件复位 LAN8720A，复位完成以后此位会自动清零。第 231~233 行等待 LAN8720A 软件复位完成，也就是判断 BMCR 的 bit15 位是否为 1，为 1 的话表示还没有复位完成。第 234 行重新向 BMCR 寄存器写入以前的值，也就是 229 行读出的那个值。

至此网络的复位引脚驱动修改完成，重新编译 uboot，然后将 u-boot.bin 烧写到 SD 卡中并启动，uboot 启动信息如图 33.2.7.3 所示：

```
U-Boot 2016.03 (May 13 2019 - 16:33:05 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 47C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT7016 (1024x600)
Video: 1024x600x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
Net:   FEC1
Normal Boot
Hit any key to stop autoboot:  0
```

图 33.2.7.3 uboot 启动信息

从图 33.2.6.4 中可以看到“Net： FEC1”这一行，提示当前使用的 FEC1 这个网口，也就是 ENET2。在 uboot 中使用网络之前要先设置几个环境变量，命令如下：

```
setenv ipaddr 192.168.1.55              //开发板 IP 地址
setenv ethaddr 00:04:9f:04:d2:35        //开发板网卡 MAC 地址
setenv gatewayip 192.168.1.1            //开发板默认网关
setenv netmask 255.255.255.0            //开发板子网掩码
setenv serverip 192.168.1.250           //服务器地址，也就是 Ubuntu 地址
saveenv                                 //保存环境变量
```

设置好环境变量以后就可以在 uboot 中使用网络了，用网线将 I.MX6U-ALPHA 上的 ENET2 与电脑或者路由器连接起来，保证开发板和电脑在同一个网段内，通过 ping 命令来测试一下网络连接，命令如下：

```
ping 192.168.1.250
```

结果如图 33.2.7.4 所示：

```
=> ping 192.168.1.250
FEC1 Waiting for PHY auto negotiation to complete.... done
Using FEC1 device
host 192.168.1.250 is alive
=>
```

图 33.2.7.4 ping 命令测试

从图 33.2.7.4 可以看出，有“host 192.168.1.250 is alive”这句，说明 ping 主机成功，说明 ENET2 网络工作正常。再来测试一下 ENET1 的网络是否正常工作，打开 mx6ull_alientek_emmc.h，将 CONFIG_FEC_ENET_DEV 改为 0，然后重新编译一下 uboot 并烧写到 SD 卡中重启。重启开发板，uboot 输出信息如图 33.2.7.5 所示：

```
U-Boot 2016.03 (May 13 2019 - 16:53:48 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 50C
Reset cause: POR
Board: MX6ULL 14x14 EVK
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT7016 (1024x600)
Video: 1024x600x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
Net:   FEC0
Normal Boot
Hit any key to stop autoboot:  0
=>
```

图 33.2.7.5 uboot 启动信息

从图 33.2.7.5 可以出，有“Net：FEC0”这一行，说明当前使用的 FEC0 这个网卡，也就是 ENET1，同样的 ping 一下主机，结果如图 33.2.7.5 所示：

```
=> ping 192.168.1.250
FEC0 Waiting for PHY auto negotiation to complete.... done
Using FEC0 device
host 192.168.1.250 is alive
=>
```

图 33.2.7.6 ping 命令测试

从图 33.2.7.6 可以看出，ping 主机也成功，说明 ENET1 网络也工作正常，至此，I.MX6U-ALPHA 开发板的两个网络都工作正常了，建议大家将 ENET2 设置为 uboot 的默认网卡！也就是将宏 CONFIG_FEC_ENET_DEV 设置为 1。

33.2.8 其他需要修改的地方

在 uboot 启动信息中会有“Board: MX6ULL 14x14 EVK”这一句，也就是说板子名字为“MX6ULL 14x14 EVK”，要将其改为我们所使用的板子名字，比如“MX6ULL ALIENTEK EMMC”或者“MX6ULL ALIENTEK NAND”。打开文件 mx6ull_alientek_emmc.c，找到函数 checkboard，将其改为如下所示内容：

示例代码 33.2.8.1 修改后的 checkboard 函数

```c
int checkboard(void)
{
    if (is_mx6ull_9x9_evk())
        puts("Board: MX6ULL 9x9 EVK\n");
    else
        puts("Board: MX6ULL ALIENTEK EMMC\n");

    return 0;
}
```

修改完成以后重新编译 uboot 并烧写到 SD 卡中验证，uboot 启动信息如图 33.2.8.1 所示：

```
U-Boot 2016.03 (May 21 2019 - 10:42:19 +0800)

CPU:   Freescale i.MX6ULL rev1.1 528 MHz (running at 396 MHz)
CPU:   Industrial temperature grade (-40C to 105C) at 54C
Reset cause: POR
Board: MX6ULL ALIENTEK EMMC
I2C:   ready
DRAM:  512 MiB
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Display: TFT7016 (1024x600)
Video: 1024x600x24
In:    serial
Out:   serial
Err:   serial
switch to partitions #0, OK
mmc0 is current device
Net:   FEC1
Normal Boot
Hit any key to stop autoboot:  0
=>
```

图 33.2.8.1 uboot 启动信息

从图 33.2.8.1 可以看出，Board 变成了“MX6ULL ALIENTEK EMMC”。至此 uboot 的驱动部分就修改完成了，uboot 移植也完成了，uboot 的最终目的就是启动 Linux 内核，所以需要通过启动 Linux 内核来判断 uboot 移植是否成功。在启动 Linux 内核之前我们先来学习两个重要的环境变量 bootcmd 和 bootargs。

33.3 bootcmd 和 bootargs 环境变量

uboot 中有两个非常重要的环境变量 bootcmd 和 bootargs，接下来看一下这两个环境变量。bootcmd 和 bootagrs 是采用类似 shell 脚本语言编写的，里面有很多的变量引用，这些变量其实都是环境变量，有很多是 NXP 自己定义的。文件 mx6ull_alientek_emmc.h 中的宏 CONFIG_EXTRA_ENV_SETTINGS 保存着这些环境变量的默认值，内容如下：

示例代码 33.3.1.1 宏 CONFIG_EXTRA_ENV_SETTINGS 默认值

```
113 #if defined(CONFIG_SYS_BOOT_NAND)
114 #define CONFIG_EXTRA_ENV_SETTINGS \
115     CONFIG_MFG_ENV_SETTINGS \
116     "panel=TFT43AB\0" \
117     "fdt_addr=0x83000000\0" \
118     "fdt_high=0xffffffff\0"   \
......
126         "bootz ${loadaddr} - ${fdt_addr}\0"
127 
128 #else
129 #define CONFIG_EXTRA_ENV_SETTINGS \
130     CONFIG_MFG_ENV_SETTINGS \
131     "script=boot.scr\0" \
132     "image=zImage\0" \
133     "console=ttymxc0\0" \
134     "fdt_high=0xffffffff\0" \
135     "initrd_high=0xffffffff\0" \
```

```
136     "fdt_file=undefined\0" \
......
194      "findfdt="\
195        "if test $fdt_file = undefined; then " \
196         "if test $board_name = EVK && test $board_rev = 9X9; then " \
197          "setenv fdt_file imx6ull-9x9-evk.dtb; fi; " \
198      "if test $board_name = EVK && test $board_rev = 14X14; then " \
199                     "setenv fdt_file imx6ull-14x14-evk.dtb; fi; " \
200             "if test $fdt_file = undefined; then " \
201             "echo WARNING: Could not determine dtb to use; fi; " \
202             "fi;\0" \
```

宏 CONFIG_EXTRA_ENV_SETTINGS 是个条件编译语句，使用 NAND 和 EMMC 的时候宏 CONFIG_EXTRA_ENV_SETTINGS 的值是不同的。

33.3.1 环境变量 bootcmd

bootcmd 在前面已经说了很多次了，bootcmd 保存着 uboot 默认命令，uboot 倒计时结束以后就会执行 bootcmd 中的命令。这些命令一般都是用来启动 Linux 内核的，比如读取 EMMC 或者 NAND Flash 中的 Linux 内核镜像文件和设备树文件到 DRAM 中，然后启动 Linux 内核。可以在 uboot 启动以后进入命令行设置 bootcmd 环境变量的值。如果 EMMC 或者 NAND 中没有保存 bootcmd 的值，那么 uboot 就会使用默认的值，板子第一次运行 uboot 的时候都会使用默认值来设置 bootcmd 环境变量。打开文件 include/env_default.h，在此文件中有如下所示内容：

示例代码 33.3.1.1 默认环境变量

```
14 env_t environment __PPCENV__ = {
15     ENV_CRC,    /* CRC Sum */
16 #ifdef CONFIG_SYS_REDUNDAND_ENVIRONMENT
17     1,      /* Flags: valid */
18 #endif
19     {
20 #elif defined(DEFAULT_ENV_INSTANCE_STATIC)
21 static char default_environment[] = {
22 #else
23 const uchar default_environment[] = {
24 #endif
25 #ifdef  CONFIG_ENV_CALLBACK_LIST_DEFAULT
26     ENV_CALLBACK_VAR "=" CONFIG_ENV_CALLBACK_LIST_DEFAULT "\0"
27 #endif
28 #ifdef  CONFIG_ENV_FLAGS_LIST_DEFAULT
29     ENV_FLAGS_VAR "=" CONFIG_ENV_FLAGS_LIST_DEFAULT "\0"
30 #endif
31 #ifdef  CONFIG_BOOTARGS
32     "bootargs=" CONFIG_BOOTARGS        "\0"
33 #endif
34 #ifdef  CONFIG_BOOTCOMMAND
```

```
35      "bootcmd="  CONFIG_BOOTCOMMAND      "\0"
36  #endif
37  #ifdef  CONFIG_RAMBOOTCOMMAND
38      "ramboot="  CONFIG_RAMBOOTCOMMAND       "\0"
39  #endif
40  #ifdef  CONFIG_NFSBOOTCOMMAND
41      "nfsboot="  CONFIG_NFSBOOTCOMMAND       "\0"
42  #endif
43  #if defined(CONFIG_BOOTDELAY) && (CONFIG_BOOTDELAY >= 0)
44      "bootdelay="    __stringify(CONFIG_BOOTDELAY)   "\0"
45  #endif
46  #if defined(CONFIG_BAUDRATE) && (CONFIG_BAUDRATE >= 0)
47      "baudrate=" __stringify(CONFIG_BAUDRATE)    "\0"
48  #endif
49  #ifdef  CONFIG_LOADS_ECHO
50      "loads_echo="   __stringify(CONFIG_LOADS_ECHO)  "\0"
51  #endif
52  #ifdef  CONFIG_ETHPRIME
53      "ethprime=" CONFIG_ETHPRIME         "\0"
54  #endif
55  #ifdef  CONFIG_IPADDR
56      "ipaddr="   __stringify(CONFIG_IPADDR)  "\0"
57  #endif
58  #ifdef  CONFIG_SERVERIP
59      "serverip=" __stringify(CONFIG_SERVERIP)    "\0"
60  #endif
61  #ifdef  CONFIG_SYS_AUTOLOAD
62      "autoload=" CONFIG_SYS_AUTOLOAD     "\0"
63  #endif
64  #ifdef  CONFIG_PREBOOT
65      "preboot="  CONFIG_PREBOOT          "\0"
66  #endif
67  #ifdef  CONFIG_ROOTPATH
68      "rootpath=" CONFIG_ROOTPATH         "\0"
69  #endif
70  #ifdef  CONFIG_GATEWAYIP
71      "gatewayip="    __stringify(CONFIG_GATEWAYIP)   "\0"
72  #endif
73  #ifdef  CONFIG_NETMASK
74      "netmask="  __stringify(CONFIG_NETMASK) "\0"
75  #endif
76  #ifdef  CONFIG_HOSTNAME
77      "hostname=" __stringify(CONFIG_HOSTNAME)    "\0"
```

```
78 #endif
79 #ifdef  CONFIG_BOOTFILE
80     "bootfile=" CONFIG_BOOTFILE        "\0"
81 #endif
82 #ifdef  CONFIG_LOADADDR
83     "loadaddr=" __stringify(CONFIG_LOADADDR)    "\0"
84 #endif
85 #ifdef  CONFIG_CLOCKS_IN_MHZ
86     "clocks_in_mhz=1\0"
87 #endif
88 #if defined(CONFIG_PCI_BOOTDELAY) && (CONFIG_PCI_BOOTDELAY > 0)
89     "pcidelay=" __stringify(CONFIG_PCI_BOOTDELAY)"\0"
90 #endif
91 #ifdef  CONFIG_ENV_VARS_UBOOT_CONFIG
92     "arch="     CONFIG_SYS_ARCH         "\0"
93     "cpu="      CONFIG_SYS_CPU          "\0"
94     "board="    CONFIG_SYS_BOARD        "\0"
95     "board_name="   CONFIG_SYS_BOARD        "\0"
96 #ifdef CONFIG_SYS_VENDOR
97     "vendor="   CONFIG_SYS_VENDOR       "\0"
98 #endif
99 #ifdef CONFIG_SYS_SOC
100     "soc="      CONFIG_SYS_SOC          "\0"
101 #endif
102 #endif
103 #ifdef  CONFIG_EXTRA_ENV_SETTINGS
104     CONFIG_EXTRA_ENV_SETTINGS
105 #endif
106     "\0"
107 #ifdef DEFAULT_ENV_INSTANCE_EMBEDDED
108     }
109 #endif
110 };
```

environment 是个 env_t 类型的变量，env_t 类型如下：

示例代码 33.3.1.2 env_t 结构体

```
156 typedef struct environment_s {
157     uint32_t    crc;        /* CRC32 over data bytes    */
158 #ifdef CONFIG_SYS_REDUNDAND_ENVIRONMENT
159     unsigned char   flags;      /* active/obsolete flags    */
160 #endif
161     unsigned char   data[ENV_SIZE]; /* Environment data     */
162 } env_t
```

env_t 结构体中的 crc 为 CRC 值，flags 是标志位，data 数组就是环境变量值。因此，environment

就是用来保存默认环境变量的，在示例代码 33.3.1.1 中指定了很多环境变量的默认值，比如 bootcmd 的默认值就是 CONFIG_BOOTCOMMAND，bootargs 的默认值就是 CONFIG_BOOTARGS。我们可以在 mx6ull_alientek_emmc.h 文件中通过设置宏 CONFIG_BOOTCOMMAND 来设置 bootcmd 的默认值，NXP 官方设置的 CONFIG_BOOTCOMMAND 值如下：

示例代码 33.3.1.3 CONFIG_BOOTCOMMAND 默认值

```
204 #define CONFIG_BOOTCOMMAND \
205        "run findfdt;" \
206        "mmc dev ${mmcdev};" \
207        "mmc dev ${mmcdev}; if mmc rescan; then " \
208            "if run loadbootscript; then " \
209                "run bootscript; " \
210            "else " \
211                "if run loadimage; then " \
212                    "run mmcboot; " \
213                "else run netboot; " \
214                "fi; " \
215            "fi; " \
216        "else run netboot; fi"
```

看起来很复杂的样子！因为 uboot 使用了类似 shell 脚本语言的方式来编写的，我们一行一行来分析。

第 205 行，run findfdt；使用的是 uboot 的 run 命令来运行 findfdt，findfdt 是 NXP 自行添加的环境变量。findfdt 是用来查找开发板对应的设备树文件(.dtb)。IMX6ULL EVK 的设备树文件为 imx6ull-14x14-evk.dtb，findfdt 内容如下：

```
"findfdt="\
"if test $fdt_file = undefined; then " \
"if test $board_name = EVK && test $board_rev = 9X9; then " \
        "setenv fdt_file imx6ull-9x9-evk.dtb; fi; " \
    "if test $board_name = EVK && test $board_rev = 14X14; then " \
        "setenv fdt_file imx6ull-14x14-evk.dtb; fi; " \
    "if test $fdt_file = undefined; then " \
        "echo WARNING: Could not determine dtb to use; fi; " \
"fi;\0" \
```

findfdt 里面用到的变量有 fdt_file，board_name，board_rev，这三个变量内容如下：

```
fdt_file=undefined，board_name=EVK，board_rev=14X14
```

findfdt 做的事情就是判断，fdt_file 是否为 undefined，如果 fdt_file 为 undefined 的话那就要根据板子信息得出所需的.dtb 文件名。此时 fdt_file 为 undefined，所以根据 board_name 和 board_rev 来判断实际所需的.dtb 文件，如果 board_name 为 EVK 并且 board_rev=9x9 的话 fdt_file 就为 imx6ull-9x9-evk.dtb。如果 board_name 为 EVK 并且 board_rev=14x14 的话 fdt_file 就设置为 imx6ull-14x14-evk.dtb。因此 IMX6ULL EVK 板子的设备树文件就是 imx6ull-14x14-evk.dtb，

因此 run findfdt 的结果就是设置 fdt_file 为 imx6ull-14x14-evk.dtb。

第 206 行，mmc dev ${mmcdev}用于切换 mmc 设备，mmcdev 为 1，因此这行代码就是：mmc dev 1，也就是切换到 EMMC 上。

第 207 行，先执行 mmc dev ${mmcdev}切换到 EMMC 上，然后使用命令 mmc rescan 扫描看有没有 SD 卡或者 EMMC 存在，如果没有的话就直接跳到 216 行，执行 run netboot，netboot 也是一个自定义的环境变量，这个变量是从网络启动 Linux 的。如果 mmc 设备存在的话就从 mmc 设备启动。

第 208 行，运行 loadbootscript 环境变量，此环境变量内容如下：

```
loadbootscript=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${script};
```

其中 mmcdev=1，mmcpart=1，loadaddr=0x80800000，script= boot.scr，因此展开以后就是：

```
loadbootscript=fatload mmc 1:1 0x80800000 boot.scr;
```

loadbootscript 就是从 mmc1 的分区 1 中读取文件 boot.src 到 DRAM 的 0X80800000 处。但是 mmc1 的分区 1 中没有 boot.src 这个文件，可以使用命令“ls mmc 1:1”查看一下 mmc1 分区 1 中的所有文件，看看有没有 boot.src 这个文件。

第 209 行，如果加载 boot.src 文件成功的话就运行 bootscript 环境变量，bootscript 的内容如下：

```
bootscript=echo Running bootscript from mmc ...;
source
```

因为 boot.src 文件不存在，所以 bootscript 也就不会运行。

第 211 行，如果 loadbootscript 没有找到 boot.src 的话就运行环境变量 loadimage，环境变量 loadimage 内容如下：

```
loadimage=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${image}
```

其中 mmcdev=1，mmcpart=1，loadaddr=0x80800000，image = zImage，展开以后就是：

```
loadimage=fatload mmc 1:1 0x80800000 zImage
```

可以看出 loadimage 就是从 mmc1 的分区中读取 zImage 到内存的 0X80800000 处，而 mmc1 的分区 1 中存在 zImage。

第 212 行，加载 linux 镜像文件 zImage 成功以后就运行环境变量 mmcboot，否则的话运行 netboot 环境变量。mmcboot 环境变量如下：

示例代码 33.3.1.4 mmcboot 环境变量

```
154 "mmcboot=echo Booting from mmc ...; " \
155     "run mmcargs; " \
156     "if test ${boot_fdt} = yes || test ${boot_fdt} = try; then " \
157         "if run loadfdt; then " \
158             "bootz ${loadaddr} - ${fdt_addr}; " \
159         "else " \
160             "if test ${boot_fdt} = try; then " \
161                 "bootz; " \
162             "else " \
163                 "echo WARN: Cannot load the DT; " \
164             "fi; " \
165         "fi; " \
166     "else " \
167         "bootz; " \
168     "fi;\0" \
```

第 154 行，输出信息“Booting from mmc ...”。

第 155 行，运行环境变量 mmcargs，mmcargs 用来设置 bootargs，后面分析 bootargs 的时候

在学习。

第156行，判断boot_fdt是否为yes或者try，根据uboot输出的环境变量信息可知boot_fdt=try。因此会执行157行的语句。

第157行，运行环境变量loadfdt，环境变量loadfdt定义如下：

```
loadfdt=fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr} ${fdt_file}
```

展开以后就是：

```
loadfdt=fatload mmc 1:1 0x83000000 imx6ull-14x14-evk.dtb
```

因此loadfdt的作用就是从mmc1的分区1中读取imx6ull-14x14-evk.dtb文件并放到0x83000000处。

第158行，如果读取.dtb文件成功的话那就调用命令bootz启动linux，调用方法如下：

```
bootz ${loadaddr} - ${fdt_addr};
```

展开就是：

```
bootz 0x80800000 - 0x83000000 (注意 '-' 前后要有空格)
```

至此Linux内核启动，如此复杂的设置就是为了从EMMC中读取zImage镜像文件和设备树文件。经过分析，浓缩出来的仅仅是4行精华：

```
mmc dev 1                                         //切换到 EMMC
fatload mmc 1:1 0x80800000 zImage                 //读取 zImage 到 0x80800000 处
fatload mmc 1:1 0x83000000 imx6ull-14x14-evk.dtb  //读取设备树到 0x83000000 处
bootz 0x80800000 - 0x83000000                     //启动 Linux
```

NXP官方将CONFIG_BOOTCOMMAND写的这么复杂只有一个目的：为了兼容多个板子，所以写了个很复杂的脚本。当我们明确知道我们所使用的板子的时候就可以大幅简化宏CONFIG_BOOTCOMMAND的设置，比如我们要从EMMC启动，那么宏CONFIG_BOOTCOMMAND就可简化为：

```
#define CONFIG_BOOTCOMMAND \
        "mmc dev 1;" \
        "fatload mmc 1:1 0x80800000 zImage;" \
        "fatload mmc 1:1 0x83000000 imx6ull-alientek-emmc.dtb;" \
        "bootz 0x80800000 - 0x83000000;"
```

或者可以直接在uboot中设置bootcmd的值，这个值就是保存到EMMC中的，命令如下：

```
setenv bootcmd 'mmc dev 1; fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000;'
```

33.3.2 环境变量 bootargs

bootargs保存着uboot传递给Linux内核的参数，在上一小节讲解bootcmd的时候说过，bootargs环境变量是由mmcargs设置的，mmcargs环境变量如下：

```
mmcargs=setenv bootargs console=${console},${baudrate} root=${mmcroot}
```

其中console=ttymxc0，baudrate=115200，mmcroot=/dev/mmcblk1p2 rootwait rw，因此将mmcargs展开以后就是：

```
mmcargs=setenv bootargs console= ttymxc0, 115200 root= /dev/mmcblk1p2 rootwait rw
```

可以看出环境变量mmcargs就是设置bootargs的值为"console= ttymxc0, 115200 root= /dev/mmcblk1p2 rootwait rw"，bootargs就是设置了很多的参数的值，这些参数Linux内核会使用到，常用的参数有：

1、console

console 用来设置 linux 终端(或者叫控制台)，也就是通过什么设备来和 Linux 进行交互，是串口还是 LCD 屏幕？如果是串口的话应该是串口几等等。一般设置串口作为 Linux 终端，这样我们就可以在电脑上通过 SecureCRT 来和 linux 交互了。这里设置 console 为 ttymxc0，因为 linux 启动以后 I.MX6ULL 的串口 1 在 linux 下的设备文件就是/dev/ttymxc0，在 Linux 下，一切皆文件。

ttymxc0 后面有个“,115200”，这是设置串口的波特率，console=ttymxc0,115200 综合起来就是设置 ttymxc0（也就是串口 1）作为 Linux 的终端，并且串口波特率设置为 115200。

2、root

root 用来设置根文件系统的位置，root=/dev/mmcblk1p2 用于指明根文件系统存放在 mmcblk1 设备的分区 2 中。EMMC 版本的核心板启动 linux 以后会存在/dev/mmcblk0、/dev/mmcblk1、/dev/mmcblk0p1、/dev/mmcblk0p2、/dev/mmcblk1p1 和/dev/mmcblk1p2 这样的文件，其中/dev/mmcblkx(x=0~n)表示 mmc 设备，而/dev/mmcblkxpy(x=0~n,y=1~n)表示 mmc 设备 x 的分区 y。在 I.MX6U-ALPHA 开发板中/dev/mmcblk1 表示 EMMC，而/dev/mmcblk1p2 表示 EMMC 的分区 2。

root 后面有“rootwait rw”，rootwait 表示等待 mmc 设备初始化完成以后再挂载，否则的话 mmc 设备还没初始化完成就挂载根文件系统会出错的。rw 表示根文件系统是可以读写的，不加 rw 的话可能无法在根文件系统中进行写操作，只能进行读操作。

3、rootfstype

此选项一般配置 root 一起使用，rootfstype 用于指定根文件系统类型，如果根文件系统为 ext 格式的话此选项无所谓。如果根文件系统是 yaffs、jffs 或 ubifs 的话就需要设置此选项，指定根文件系统的类型。

bootargs 常设置的选项就这三个，后面遇到其他选项的话再讲解。

33.4 uboot 启动 Linux 测试

uboot 已经移植好了，bootcmd 和 bootargs 这两个重要的环境变量也讲解了，接下来就要测试一下 uboot 能不能完成它的工作：启动 Linux 内核。我们测试两种启动 Linux 内核的方法，一种是直接从 EMMC 启动，一种是从网络启动。

33.4.1 从 EMMC 启动 Linux 系统

从 EMMC 启动也就是将编译出来的 Linux 镜像文件 zImage 和设备树文件保存在 EMMC 中，uboot 从 EMMC 中读取这两个文件并启动，这个是我们产品最终的启动方式。但是我们目前还没有讲解如何移植 linux 和设备树文件，以及如何将 zImage 和设备树文件保存到 EMMC 中。不过大家拿到手的 I.MX6U-ALPHA 开发板(EMMC 版本)已经将 zImage 文件和设备树文件烧写到了 EMMC 中，所以我们可以直接读取来测试。先检查一下 EMMC 的分区 1 中有没有 zImage 文件和设备树文件，输入命令“ls mmc 1:1”，结果如图 33.4.1.1 所示：

```
=> ls mmc 1:1
  6073416   zimage
    37331   imx6ull-alientek-emmc.dtb
2 file(s), 0 dir(s)

=>
```

图 33.4.1.1 EMMC 分区 1 文件

从图 33.4.1.1 中可以看出，此时 EMMC 分区 1 中存在 zimage 和 imx6ull-alientek-emmc.dtb 这两个文件，所以我们可以测试新移植的 uboot 能不能启动 linux 内核。设置 bootargs 和 bootcmd 这两个环境变量，设置如下：

```
setenv bootargs 'console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw'
setenv bootcmd 'mmc dev 1; fatload mmc 1:1 80800000 zImage; fatload mmc 1:1 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000;'
saveenv
```

设置好以后直接输入 boot，或者 run bootcmd 即可启动 Linux 内核，如果 Linux 内核启动成功的话就会输出如图 33.4.1.2 所示的启动信息：

```
switch to partitions #0, OK
mmc1(part 0) is current device
reading zImage
6073416 bytes read in 150 ms (38.6 MiB/s)
reading imx6ull-alientek-emmc.dtb
37331 bytes read in 18 ms (2 MiB/s)
Kernel image @ 0x80800000 [ 0x000000 - 0x5cac48 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300c1d2

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-201
7.01) ) #10 SMP PREEMPT Mon Apr 22 15:21:47 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 UltraLite 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
Memory policy: Data cache writealloc
PERCPU: Embedded 12 pages/cpu @8bb31000 s17280 r8192 d23680 u49152
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 179900K/524288K available (7510K kernel code, 368K rwdata, 2596K rodata,
 412K init, 442K bss, 16708K reserved, 327680K cma-reserved, 0K highmem)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
```

图 33.4.1.2 linux 内核启动成功

33.4.2 从网络启动 Linux 系统

从网络启动 linux 系统的唯一目的就是为了调试！不管是为了调试 linux 系统还是 linux 下的驱动。每次修改 linux 系统文件或者 linux 下的某个驱动以后都要将其烧写到 EMMC 中去测试，这样太麻烦了。我们可以设置 linux 从网络启动，也就是将 linux 镜像文件和根文件系统都放到 Ubuntu 下某个指定的文件夹中，这样每次重新编译 linux 内核或者某个 linux 驱动以后只需要使用 cp 命令将其拷贝到这个指定的文件夹中即可，这样就不用需要频繁的烧写 EMMC，这样就加快了开发速度。我们可以通过 nfs 或者 tftp 从 Ubuntu 中下载 zImage 和设备树文件，根文件系统的话也可以通过 nfs 挂载，不过本小节我们不讲解如何通过 nfs 挂载根文件系统，这个在讲解根文件系统移植的时候再讲解。这里我们使用 tftp 从 Ubuntu 中下载 zImage 和设备树文件，前提是要将 zImage 和设备树文件放到 Ubuntu 下的 tftp 目录中，具体方法在 30.4.4 小节讲解 tftp 命令的时候已经详细的介绍过了。

设置 bootargs 和 bootcmd 这两个环境变量，设置如下：

```
setenv bootargs 'console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw'
```

```
setenv bootcmd 'tftp 80800000 zImage; tftp 83000000 imx6ull-alientek-emmc.dtb; bootz 80800000 - 83000000'
saveenv
```

一开始是通过 tftp 下载 zImage 和 imx6ull-alientek-emmc.dtb 这两个文件，过程如下图 33.4.1.3 所示：

```
FEC1 Waiting for PHY auto negotiation to complete.... done
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.251
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ########################
         2.3 MiB/s
done
Bytes transferred = 6073416 (5cac48 hex)
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.251
Filename 'imx6ull-alientek-emmc.dtb'.
Load address: 0x83000000
Loading: ###
         2.1 MiB/s
done
Bytes transferred = 37331 (91d3 hex)
```

图 33.4.1.3 下载过程

下载完成以后就是启动 Linux 内核，启动过程如图 33.4.1.4 所示：

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-201
7.01) ) #10 SMP PREEMPT Mon Apr 22 15:21:47 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 UltraLite 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
Memory policy: Data cache writealloc
PERCPU: Embedded 12 pages/cpu @8bb31000 s17280 r8192 d23680 u49152
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 179900K/524288K available (7510K kernel code, 368K rwdata, 2596K rodata,
 412K init, 442K bss, 16708K reserved, 327680K cma-reserved, 0K highmem)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xa0800000 - 0xff000000   (1512 MB)
    lowmem  : 0x80000000 - 0xa0000000   ( 512 MB)
    pkmap   : 0x7fe00000 - 0x80000000   (   2 MB)
    modules : 0x7f000000 - 0x7fe00000   (  14 MB)
      .text : 0x80008000 - 0x809e6c20   (10108 kB)
      .init : 0x809e7000 - 0x80a4e000   ( 412 kB)
      .data : 0x80a4e000 - 0x80aaa220   ( 369 kB)
       .bss : 0x80aad000 - 0x80b1ba24   ( 443 kB)
```

图 33.4.1.5 Linux 启动过程

uboot 移植到此结束，简单总结一下 uboot 移植的过程：

①、不管是购买的开发板还是自己做的开发板，基本都是参考半导体厂商的 dmeo 板，而半导体厂商会在他们自己的开发板上移植好 uboot、linux kernel 和 systemfs 等，最终制作好 BSP 包提供给用户。我们可以在官方提供的 BSP 包的基础上添加我们的板子，也就是俗称的移植。

②、我们购买的开发板或者自己做的板子一般都不会原封不动的照抄半导体厂商的 demo 板，

都会根据实际的情况来做修改，既然有修改就必然涉及到 uboot 下驱动的移植。

③、一般 uboot 中需要解决串口、NAND、EMMC 或 SD 卡、网络和 LCD 驱动，因为 uboot 的主要目的就是启动 Linux 内核，所以不需要考虑太多的外设驱动。

④、在 uboot 中添加自己的板子信息，根据自己板子的实际情况来修改 uboot 中的驱动。

第三十四章 U-Boot 图形化配置及其原理

在前两章中我们知道 uboot 可以通过 mx6ull_alientek_emmc_defconfig 来配置，或者通过文件 mx6ull_alientek_emmc.h 来配置 uboot。还有另外一种配置 uboot 的方法，就是图形化配置，以前的 uboot 是不支持图形化配置，只有 Linux 内核才支持图像化配置。不过不知道从什么时候开始，uboot 也支持图形化配置了，本章我们就来学习一下如何通过图形化配置 uboot，并且学习一下图形化配置的原理，因为后面学习 Linux 驱动开发的时候可能要修改图形配置文件。

34.1 U-Boot 图形化配置体验

uboot 或 Linux 内核可以通过输入“make menuconfig”来打开图形化配置界面，menuconfig 是一套图形化的配置工具，需要 ncurses 库支持。ncurses 库提供了一系列的 API 函数供调用者生成基于文本的图形界面，因此需要先在 Ubuntu 中安装 ncurses 库，命令如下：

```
sudo apt-get install build-essential
sudo apt-get install libncurses5
sudo apt-get install libncurses5-dev
```

menuconfig 重点会用到两个文件：.config 和 Kconfig，.config 文件前面已经说了，这个文件保存着 uboot 的配置项，使用 menuconfig 配置完 uboot 以后肯定要更新.config 文件。Kconfig 文件是图形界面的描述文件，也就是描述界面应该有什么内容，很多目录下都会有 Kconfig 文件。

在打开图形化配置界面之前，要先使用“make xxx_defconfig”对 uboot 进行一次默认配置，只需要一次即可。如果使用“make clean”清理了工程的话就那就需要重新使用“make xxx_defconfig”再对 uboot 进行一次配置。进入 uboot 根目录，输入如下命令：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mx6ull_alientek_emmc_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

如果已经在 uboot 的顶层 Makefile 中定义了 ARCH 和 CROSS_COMPILE 的值，那么上述命令可以简化为：

```
make mx6ull_alientek_emmc_defconfig
make menuconfig
```

打开后的图形化界面如图 34.1.1 所示：

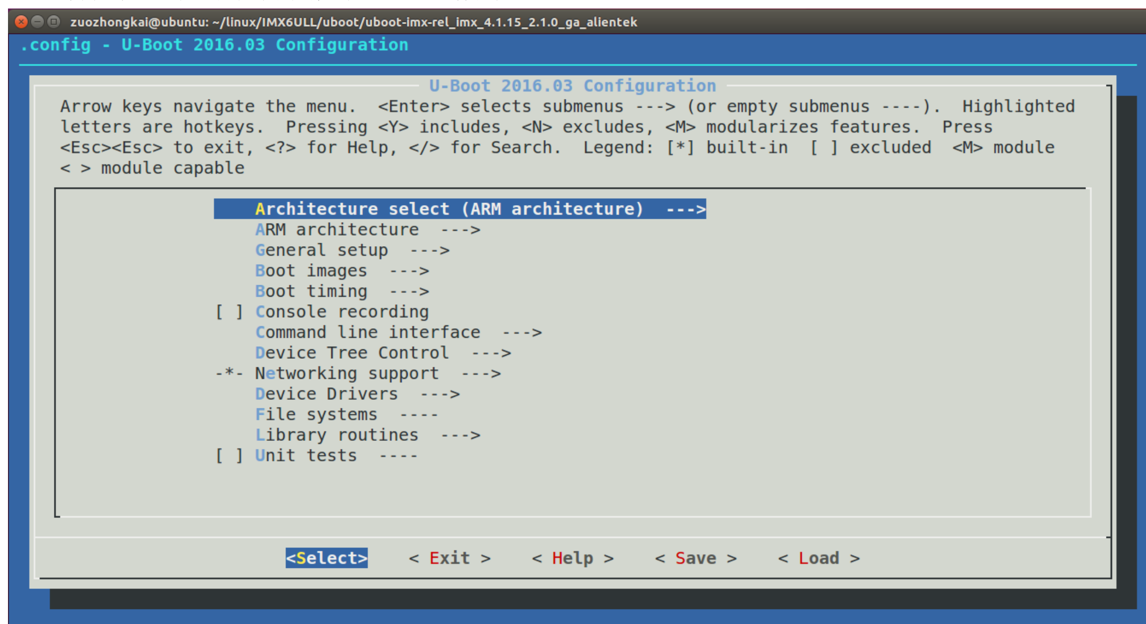


图 34.1.1 uboot 图形化配置界面

图 34.1.1 就是主界面，主界面上方的英文就是简单的操作说明，操作方法如下：

通过键盘上的“↑”和“↓”键来选择要配置的菜单，按下“Enter”键进入子菜单。菜单中高亮的字母就是此菜单的热键，在键盘上按下此高亮字母对应的键可以快速选中对应的菜单。选中子菜单以后按下“Y”键就会将相应的代码编译进 Uboot 中，菜单前面变为“<*>”。按下“N”键不编译相应的代码，按下“M”键就会将相应的代码编译为模块，菜单前面变为“<M>”。

按两下“Esc”键退出，也就是返回到上一级，按下“?”键查看此菜单的帮助信息，按下“/”键打开搜索框，可以在搜索框输入要搜索的内容。

在配置界面下方会有五个按钮，这五个按钮的功能如下：

<Select>：选中按钮，和“Enter”键的功能相同，负责选中并进入某个菜单。

<Exit>：退出按钮，和按两下“Esc”键功能相同，退出当前菜单，返回到上一级。

<Help>：帮助按钮，查看选中菜单的帮助信息。

<Save>：保存按钮，保存修改后的配置文件。

<Load>：加载按钮，加载指定的配置文件。

在图 34.1.1 中共有 13 个配置主配置项，通过键盘上的上下键调节配置项。后面跟着“--->”表示此配置项是有子配置项的，按下回车键就可以进入子配置项。

我们就以如何使能 dns 命令为例，讲解一下如何通过图形化界面来配置 uboot。进入“Command line interface --->”这个配置项，此配置项用于配置 uboot 的命令，进入以后如图 34.1.2 所示：

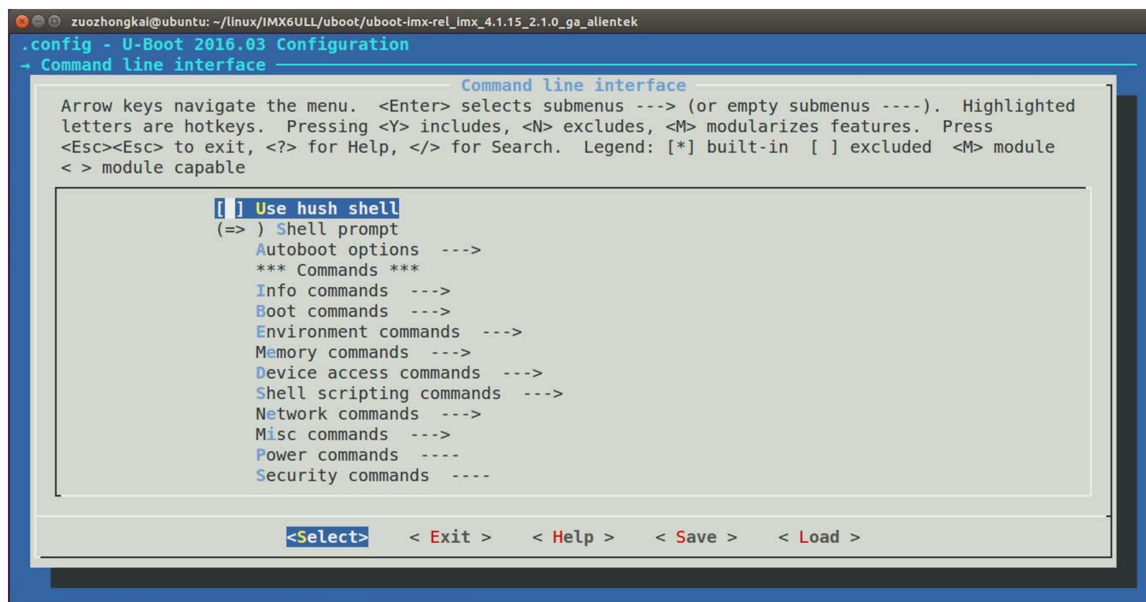


图 34.1.2 Command line interface 配置项

从图 34.1.2 可以看出，有很多配置项，这些配置项也有子配置项，选择“Network commands --->”，进入网络相关命令配置项，如图 34.1.3 所示：

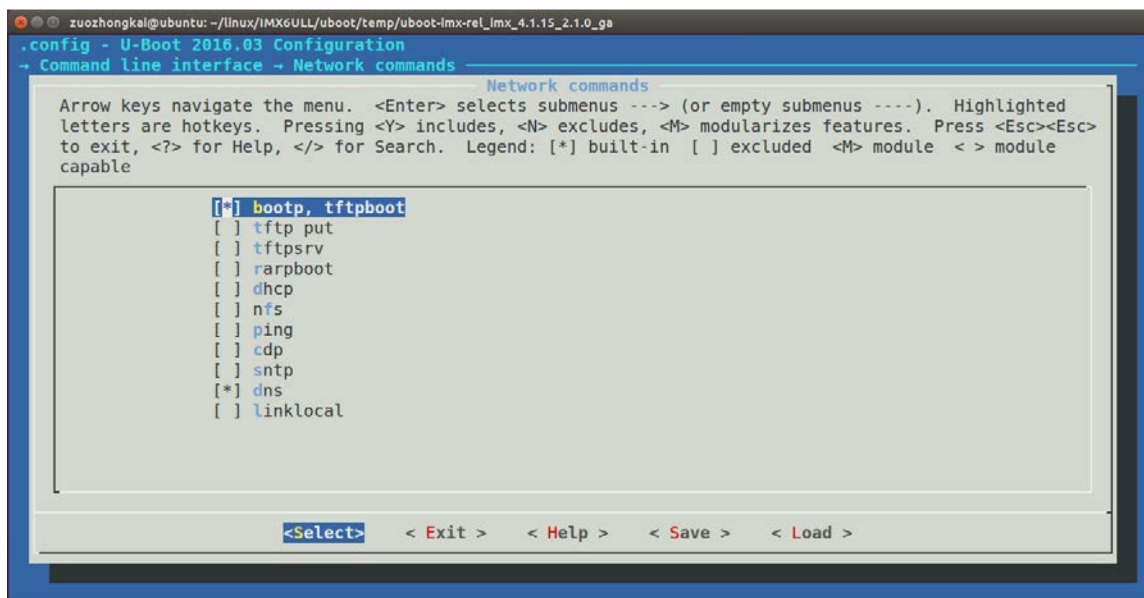


图 34.1.3 Network commands 配置项

从图 34.1.3 可以看出，uboot 中有很多和网络有关的命令，比如 bootp、tftpboot、dhcp 等等。选中 dns，然后按下键盘上的“Y”键，此时 dns 前面的“[ ]”变成了“[*]”，如图 34.1.4 所示：

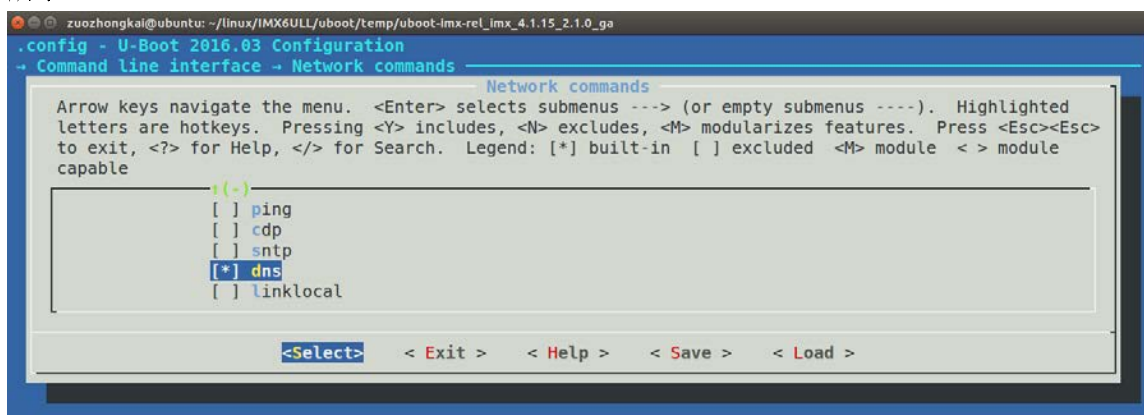


图 34.1.4 选中 dns 命令

每个选项有 3 种编译选项：编译进 uboot 中(也就是编译进 u-boot.bin 中)、取消编译(也就是不编译这个功能模块)、编译为模块。按下“Y”键表示编译进 uboot 中，此时“[ ]”变成了“[*]”；按下“N”表示不编译，“[ ]”默认表示不编译；有些功能模块是支持编译为模块的，这个一般在 Linux 内核里面很常用，uboot 下面不使用，如果要将某个功能编译为模块，那就按下“M”，此时“[ ]”就会变为“< M >”。

细心的朋友应该会发现，在 mx6ull_alientek_emmc.h 里面我们配置使能了 dhcp 和 ping 命令，但是在图 34.1.3 中 dhcp 和 ping 前面的“[ ]”并不是“[*]”，也就是说不编译 dhcp 和 ping 命令，这不是冲突了吗？实际情况是 dhcp 和 ping 命令是会编译的。之所以在图 34.1.3 中没有体现出来时因为我们是直接在 mx6ull_alientek_emmc.h 中定义的宏 CONFIG_CMD_PING 和 CONFIG_CMD_DHCP，而 menuconfig 是通过读取.config 文件来判断使能了哪些功能，.config 里面并没有宏 CONFIG_CMD_PING 和 CONFIG_CMD_DHCP，所以 menuconfig 就会识别出错。

选中 dns，然后按下“H”或者“?”键可以打开 dns 命令的提示信息，如图 34.1.5 所示：

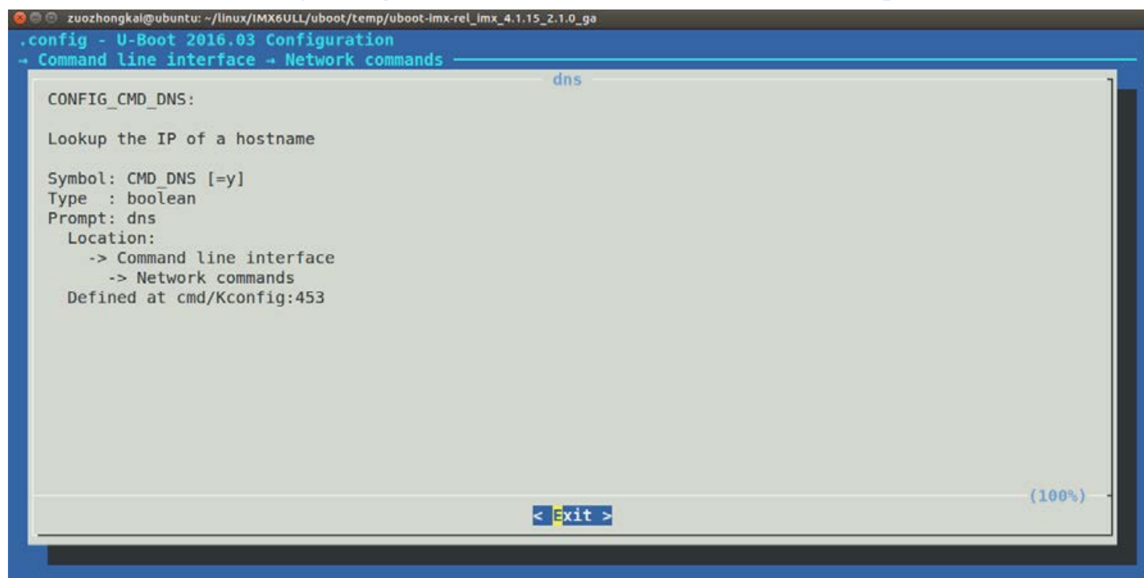


图 34.1.5 dns 命令提示信息

按两下 ESC 键即可退出提示界面，相当于返回上一层。选择 dns 命令以后，按两下 ESC 键（按两下 ESC 键相当于返回上一层），退出当前配置项，进入到上一层配置项。如果没有要修改的就按两下 ESC 键，退出到主配置界面，如果也没有其他要修改的，那就再次按两下 ESC 键退出 menuconfig 配置界面。如果修改过配置的话，在退出主界面的时候会有如图 34.1.6 所示提示：

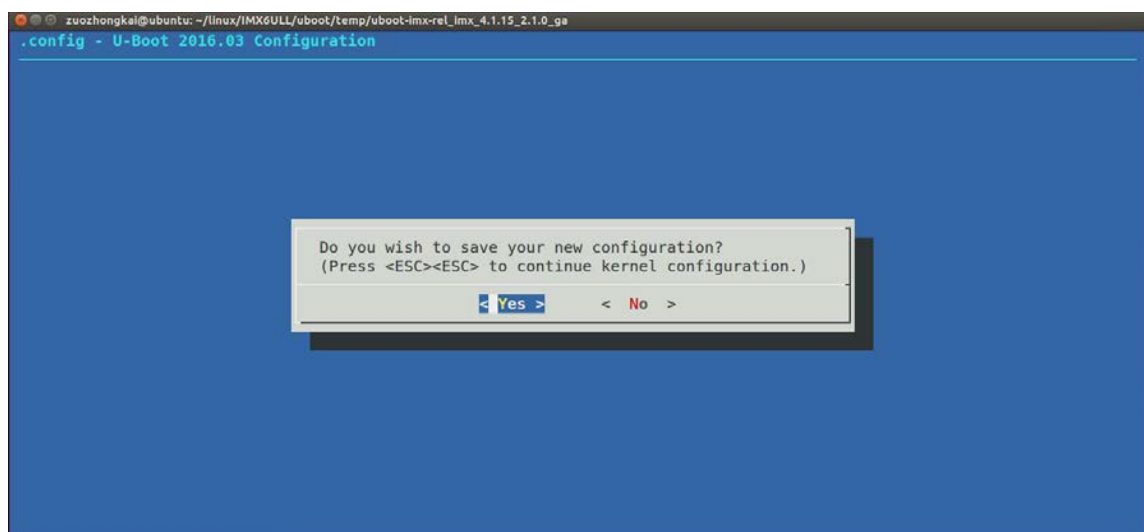


图 34.1.6 是否保存新的配置文件对话框

图 34.1.6 询问是否保存新的配置文件，通过键盘的←或→键来选择“Yes”项，然后按下键盘上的回车键确认保存。至此，我们就完成了通过图形界面使能了 uboot 的 dns 命令，打开.config 文件，会发现多了“CONFIG_CMD_DNS=y”这一行，如图 34.1.7 中的 323 行所示：

```
zuozhongkai@ubuntu: ~/linux/IMX6ULL/uboot/temp/uboot-imx-rel_imx_4.1.15_2.1.0_ga
318 # CONFIG_CMD_DHCP is not set
319 # CONFIG_CMD_NFS is not set
320 # CONFIG_CMD_PING is not set
321 # CONFIG_CMD_CDP is not set
322 # CONFIG_CMD_SNTP is not set
323 CONFIG_CMD_DNS=y
324 # CONFIG_CMD_LINK_LOCAL is not set
325
326 #
327 # Misc commands
328 #
329 # CONFIG_CMD_TIME is not set
330 CONFIG_CMD_MISC=y
331 # CONFIG_CMD_TIMER is not set
                                                        331,12        57%
```

图 34.1.7 .config 文件

使用如下命令编译 uboot：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -j16
```

千万不能使用如下命令：

```
./mx6ull_alientek_emmc.sh
```

因为 mx6ull_alientek_emmc.sh 在编译之前会清理工程，会删除掉.config 文件！通过图形化界面配置所有配置项都会被删除，结果就是竹篮打水一场空。

编译完成以后烧写到 SD 卡中，重启开发板进入 uboot 命令模式，输入"?"查看是否有"dns"命令，一般肯定有的。测试一下 dns 命令工作是否正常，使用 dns 命令来查看一下百度官网"www.baidu.com"的 IP 地址。要先设置一下 dns 服务器的 IP 地址，也就是设置环境变量 dnsip 的值，命令如下：

```
setenv dnsip 114.114.114.114
saveenv
```

设置好以后就可以使用 dns 命令查看百度官网的 IP 地址了，输入命令：

```
dns www.baidu.com
```

结果如图 34.1.8 所示：

```
=> dns www.baidu.com
FEC1 Waiting for PHY auto negotiation to complete.... done
14.215.177.38
=>
```

图 34.1.78dns 命令

从图 34.1.7 可以看出，"www.baidu.com"的 IP 地址为 14.215.177.38，说明 dns 命令工作正常。这个就是通过图形化命令来配置 uboot，一般用来使能一些命令还是很方便的，这样就不需要到处找命令的配置宏是什么，然后在到配置文件里面去定义。

34.2 menuconfig 图形化配置原理

34.2.1 make menuconfig 过程分析

当输入 make menuconfig 以后会匹配到顶层 Makefile 的如下代码：

示例代码 34.2.1.1 顶层 Makefile 代码段

```
489 %config: scripts_basic outputmakefile FORCE
490     $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

这个在 31.3.13 小节已经详细的讲解过了，其中 build=-f ./scripts/Makefile.build obj，将 490 行的规则展开就是：

```
@ make -f ./scripts/Makefile.build obj=scripts/kconfig menuconfig
```

Makefile.build 会读取 scripts/kconfig/Makefile 中的内容，在 scripts/kconfig/Makefile 中可以找到如下代码：

示例代码 34.2.1.2 scripts/kconfig/Makefile 代码段

```
36 menuconfig: $(obj)/mconf
37     $< $(silent) $(Kconfig)
```

其中 obj= scripts/kconfig，silent 是设置静默编译的，在这里可以忽略不计，Kconfig=Kconfig，因此扩展以后就是：

```
menuconfig: scripts/kconfig/mconf
  scripts/kconfig/mconf Kconfig
```

目标 menuconfig 依赖 scripts/kconfig/mconf，因此 scripts/kconfig/mconf.c 这个文件会被编译，生成 mconf 这个可执行文件。目标 menuconfig 对应的规则为 scripts/kconfig/mconf Kconfig，也就是说 mconf 会调用 uboot 根目录下的 Kconfig 文件开始构建图形配置界面。

34.2.2 Kconfig 语法简介

上一小节我们已经知道了 scripts/kconfig/mconf 会调用 uboot 根目录下的 Kconfig 文件开始构建图形化配置界面，接下来简单学习一下 Kconfig 的语法。因为后面学习 Linux 驱动开发的时候可能会涉及到修改 Kconfig，对于 Kconfig 语法我们不需要太深入的去研究，关于 Kconfig 的详细语法介绍，可以参考 linux 内核源码(不知为何 uboot 源码中没有这个文件)中的文件 Documentation/kbuild/kconfig-language.txt，本节我们大概了解其原理即可。打开 uboot 根目录下的 Kconfig，这个 Kconfig 文件就是顶层 Kconfig，我们就以这个文件为例来简单学习一下 Kconfig 语法。

1、mainmenu

故名思议 mainmenu 就是主菜单，也就是输入“make menuconfig”以后打开的默认界面，在顶层 Kconfig 中有如下代码：

示例代码 34.2.2.1 顶层 Kconfig 代码段

```
5 mainmenu "U-Boot $UBOOTVERSION Configuration"
```

上述代码就是定义了一个名为“U-Boot $UBOOTVERSION Configuration”的主菜单，其中 UBOOTVERSION=2016.03，因此主菜单名为“U-Boot 2016.03 Configuration”，如图 34.2.2.1 所示：

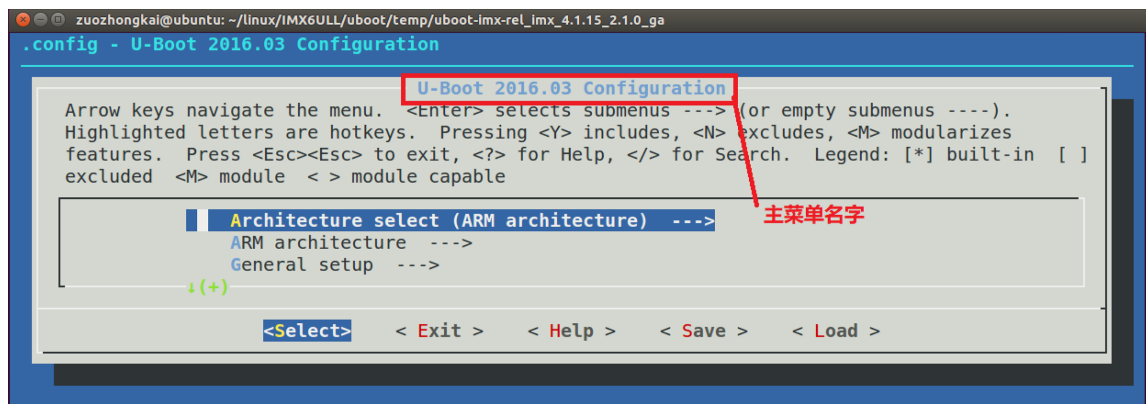


图 34.2.2.1 主菜单名字

2、调用其他目录下的 Kconfig 文件

和 makefile 一样，Kconfig 也可以调用其他子目录中的 Kconfig 文件，调用方法如下：

```
source "xxx/Kconfig"    //xxx 为具体的目录名，相对路径
```

在顶层 Kconfig 中有如下代码：

示例代码 34.2.2.2 顶层 Kconfig 代码段

```
12  source "arch/Kconfig"
......
225 
226 source "common/Kconfig"
227 
228 source "cmd/Kconfig"
229 
230 source "dts/Kconfig"
231 
232 source "net/Kconfig"
233 
234 source "drivers/Kconfig"
235 
236 source "fs/Kconfig"
237 
238 source "lib/Kconfig"
239 
240 source "test/Kconfig"
```

从示例代码 34.2.2.2 中可以看出，顶层 Kconfig 文件调用了很多其他子目录下的 Kcofig 文件，这些子目录下的 Kconfig 文件在主菜单中生成各自的菜单项。

3、menu/endmenu 条目

menu 用于生成菜单，endmenu 就是菜单结束标志，这两个一般是成对出现的。在顶层 Kconfig 中有如下代码：

示例代码 34.2.2.3 顶层 Kconfig 代码段

```
14  menu "General setup"
15 
16  config LOCALVERSION
17      string "Local version - append to U-Boot release"
18      help
19        Append an extra string to the end of your U-Boot version.
20        This will show up on your boot log, for example.
21        The string you set here will be appended after the contents of
22        any files with a filename matching localversion* in your
23        object and source tree, in that order.  Your total string can
24        be a maximum of 64 characters.
25 
......
100 endmenu     # General setup
101 
102 menu "Boot images"
```

```
103
104 config SUPPORT_SPL
105     bool
106
......
224 endmenu     # Boot images
```

示例代码 34.2.2.3 中有两个 menu/endmenu 代码块，这两个代码块就是两个子菜单，第 14 行的“menu "General setup"”表示子菜单“General setup”。第 102 行的“menu "Boot images"”表示子菜单“Boot images”。体现在主菜单界面中就如图 34.2.2.2 所示：

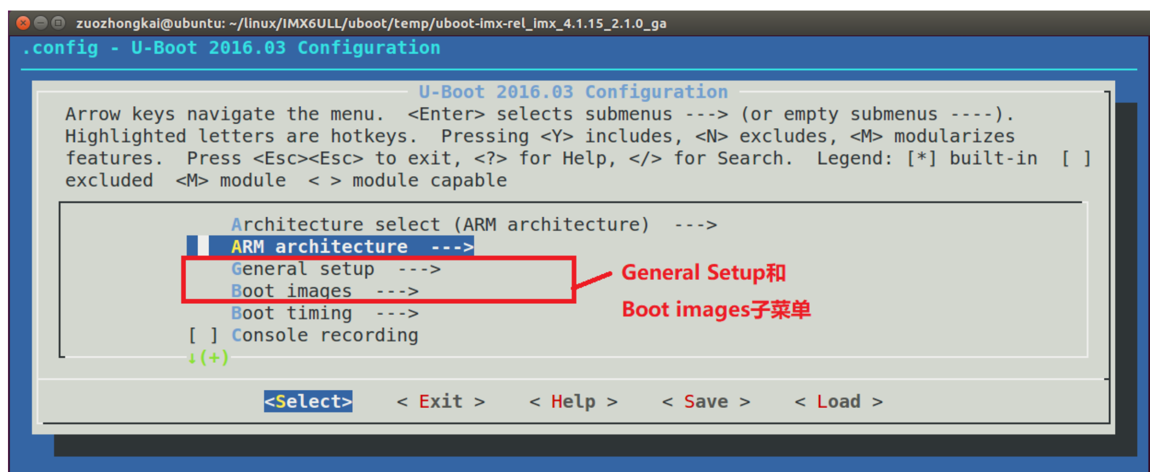


图 34.2.2.2 子菜单

在“General setup”菜单上面还有 “Architecture select (ARM architecture)”和“ARM architecture”这两个子菜单，但是在顶层 Kconfig 中并没有看到这两个子菜单对应的 menu/endmenu 代码块，那这两个子菜单是怎么来的呢？这两个子菜单就是 arch/Kconfig 文件生成的。包括主界面中的“Boot timing”、“Console recording”等等这些子菜单，都是分别由顶层 Kconfig 所调用的 common/Kconfig、cmd/Kconfig 等这些子 Kconfig 文件来创建的。

3、config 条目

顶层 Kconfig 中的“General setup”子菜单内容如下：

示例代码 34.2.2.4 顶层 Kconfig 代码段

```
14 menu "General setup"
15
16 config LOCALVERSION
17     string "Local version - append to U-Boot release"
18     help
19       Append an extra string to the end of your U-Boot version.
20       This will show up on your boot log, for example.
21       The string you set here will be appended after the contents of
22       any files with a filename matching localversion* in your
23       object and source tree, in that order.  Your total string can
24       be a maximum of 64 characters.
25
26 config LOCALVERSION_AUTO
```

```
27      bool "Automatically append version information to the version
string"
28      default y
29      help
......
45
46  config CC_OPTIMIZE_FOR_SIZE
47      bool "Optimize for size"
48      default y
49      help
......
54
55  config SYS_MALLOC_F
56      bool "Enable malloc() pool before relocation"
57      default y if DM
58      help
......
63
64  config SYS_MALLOC_F_LEN
65      hex "Size of malloc() pool before relocation"
66      depends on SYS_MALLOC_F
67      default 0x400
68      help
......
73
74  menuconfig EXPERT
75      bool "Configure standard U-Boot features (expert users)"
76      default y
77      help
......
82
83  if EXPERT
84      config SYS_MALLOC_CLEAR_ON_INIT
85      bool "Init with zeros the memory reserved for malloc (slow)"
86      default y
87      help
......
99  endif
100 endmenu     # General setup
```

可以看出，在 menu/endmenu 代码块中有大量的“config xxxx”的代码块，也就是 config 条目。config 条目就是“General setup”菜单的具体配置项，如图 34.2.2.3 所示：

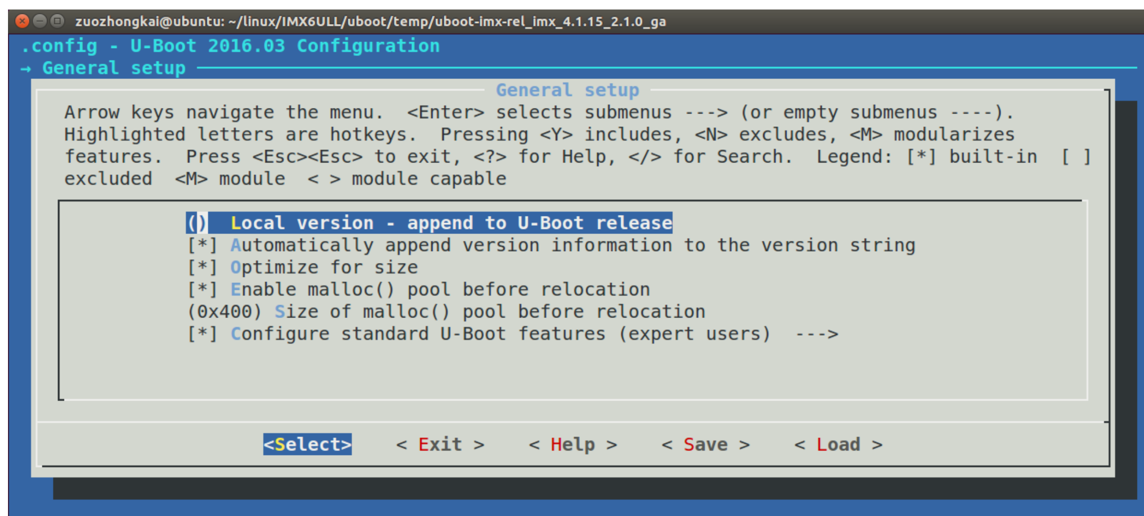


图 34.2.2.3 General setup 配置项

“config LOCALVERSION”对应着第一个配置项，“config LOCALVERSION_AUTO”对应着第二个配置项，以此类推。我们以“config LOCALVERSION”和“config LOCALVERSION_AUTO”这两个为例来分析一下 config 配置项的语法：

示例代码 34.2.2.5 顶层 Kconfig 代码段

```
16 config LOCALVERSION
17  string "Local version - append to U-Boot release"
18  help
19    Append an extra string to the end of your U-Boot version.
......
24    be a maximum of 64 characters.
25
26 config LOCALVERSION_AUTO
27  bool "Automatically append version information to the version
string"
28  default y
29  help
30    This will try to automatically determine if the current tree is a
31    release tree by looking for git tags that belong to the current
......
43
44    which is done within the script "scripts/setlocalversion".)
```

第 16 和 26 行，这两行都以 config 关键字开头，后面跟着 LOCALVERSION 和 LOCALVERSION_AUTO，这两个就是配置项名字。假如我们使能了 LOCALVERSION_AUTO 这个功能，那么就会下.config 文件中生成 CONFIG_LOCALVERSION_AUTO，这个在上一小节讲解如何使能 dns 命令的时候讲过了。由此可知，.config 文件中的“CONFIG_xxx” (xxx 就是具体的配置项名字)就是 Kconfig 文件中 config 关键字后面的配置项名字加上“CONFIG_”前缀。

config 关键字下面的这几行是配置项属性，17~24 行是 LOCALVERSION 的属性，27~44 行是 LOCALVERSION_AUTO 的属性。属性里面描述了配置项的类型、输入提示、依赖关系、帮

助信息和默认值等。

第 17 行的 string 是变量类型，也就是"CONFIG_LOCALVERSION"的变量类型。可以为：bool、tristate、string、hex 和 int，一共 5 种。最常用的是 bool、tristate 和 string 这三种，bool 类型有两种值：y 和 n，当为 y 的时候表示使能这个配置项，当为 n 的时候就禁止这个配置项。tristate 类型有三种值：y、m 和 n，其中 y 和 n 的涵义与 bool 类型一样，m 表示将这个配置项编译为模块。string 为字符串类型，所以 LOCALVERSION 是个字符串变量，用来存储本地字符串，选中以后即可输入用户定义的本地版本号，如图 34.2.2.4 所示：

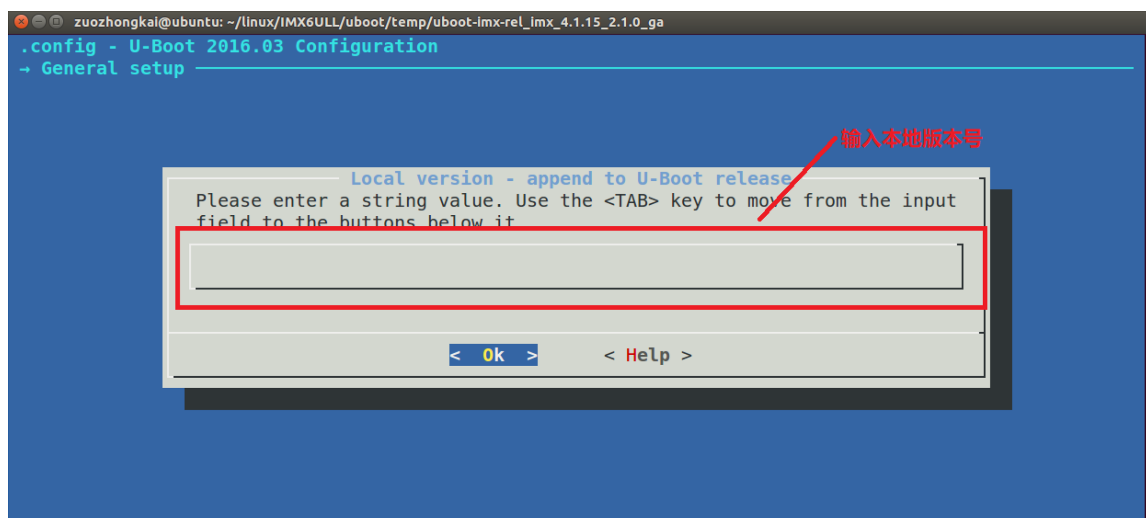


图 34.2.2.4 本地版本号配置

string 后面的"Local version - append to U-Boot release"就是这个配置项在图形界面上的显示出来的标题。

第 18 行，help 表示帮助信息，告诉我们配置项的含义，当我们按下"h"或"?"弹出来的帮助界面就是 help 的内容。

第 27 行，说明"CONFIG_LOCALVERSION_AUTO"是个 bool 类型，可以通过按下 Y 或 N 键来使能或者禁止 CONFIG_LOCALVERSION_AUTO。

第 28 行，"default y"表示 CONFIG_LOCALVERSION_AUTO 的默认值就是 y，所以这一行默认会被选中。

4、depends on 和 select

打开 arch/Kconfig 文件，在里面有这如下代码：

示例代码 34.2.2.6 arch/Kconfig 代码段

```
7  config SYS_GENERIC_BOARD
8       bool
9       depends on HAVE_GENERIC_BOARD
10
11 choice
12      prompt "Architecture select"
13      default SANDBOX
14
15 config ARC
16      bool "ARC architecture"
17      select HAVE_PRIVATE_LIBGCC
```

```
18      select HAVE_GENERIC_BOARD
19      select SYS_GENERIC_BOARD
20      select SUPPORT_OF_CONTROL
```

第 9 行，“depends on”说明“SYS_GENERIC_BOARD”项依赖于“HAVE_GENERIC_BOARD”，也就是说“HAVE_GENERIC_BOARD”被选中以后“SYS_GENERIC_BOARD”才能被选中。

第 17~20 行，“select”表示方向依赖，当选中“ARC”以后，“HAVE_PRIVATE_LIBGCC”、“HAVE_GENERIC_BOARD”、“SYS_GENERIC_BOARD”和“SUPPORT_OF_CONTROL”这四个也会被选中。

4、choice/endchoice

在 arch/Kconfig 文件中有如下代码：

示例代码 34.2.2.7 arch/Kconfig 代码段

```
11  choice
12      prompt "Architecture select"
13      default SANDBOX
14
15  config ARC
16      bool "ARC architecture"
......
21
22  config ARM
23      bool "ARM architecture"
......
29
30  config AVR32
31      bool "AVR32 architecture"
......
35
36  config BLACKFIN
37      bool "Blackfin architecture"
......
40
41  config M68K
42      bool "M68000 architecture"
......
117
118 endchoice
```

choice/endchoice 代码段定义了一组可选择项，将多个类似的配置项组合在一起，供用户单选或者多选。示例代码 34.2.2.7 就是选择处理器架构，可以从 ARC、ARM、AVR32 等这些架构中选择，这里是单选。在 uboot 图形配置界面上选择“Architecture select”，进入以后如图 34.2.2.5 所示：



图 34.2.2.5 架构选择界面

可以在图 34.2.2.5 中通过移动光标来选择所使用的 CPU 架构。第 12 行的 prompt 给出这个 choice/endchoice 段的提示信息为“Architecture select”。

5、menuconfig

menuconfig 和 menu 很类似，但是 menuconfig 是个带选项的菜单，其一般用法为：

示例代码 34.2.2.8 menuconfig 用法

```
1 menuconfig MODULES
2     bool "菜单"
3 if MODULES
4 ...
5 endif # MODULES
```

第 1 行，定义了一个可选的菜单 MODULES，只有选中了 MODULES 第 3~5 行 if 到 endif 之间的内容才会显示。在顶层 Kconfig 中有如下代码：

示例代码 34.2.2.9 顶层 Kconfig 代码段

```
14  menu "General setup"
......
74  menuconfig EXPERT
75      bool "Configure standard U-Boot features (expert users)"
76      default y
77      help
78        This option allows certain base U-Boot options and settings
79        to be disabled or tweaked. This is for specialized
80        environments which can tolerate a "non-standard" U-Boot.
81        Only use this if you really know what you are doing.
82
83  if EXPERT
84      config SYS_MALLOC_CLEAR_ON_INIT
85      bool "Init with zeros the memory reserved for malloc (slow)"
86      default y
87      help
```

```
88         This setting is enabled by default. The reserved malloc
89         memory is initialized with zeros, so first malloc calls
......
98         should be replaced by calloc - if expects zeroed memory.
99  endif
100 endmenu     # General setup
```

第 74~99 行使用 menuconfig 实现了一个菜单，路径如下：

```
General setup
  -> Configure standard U-Boot features (expert users)   --->
```

如图 34.2.2.6 所示：

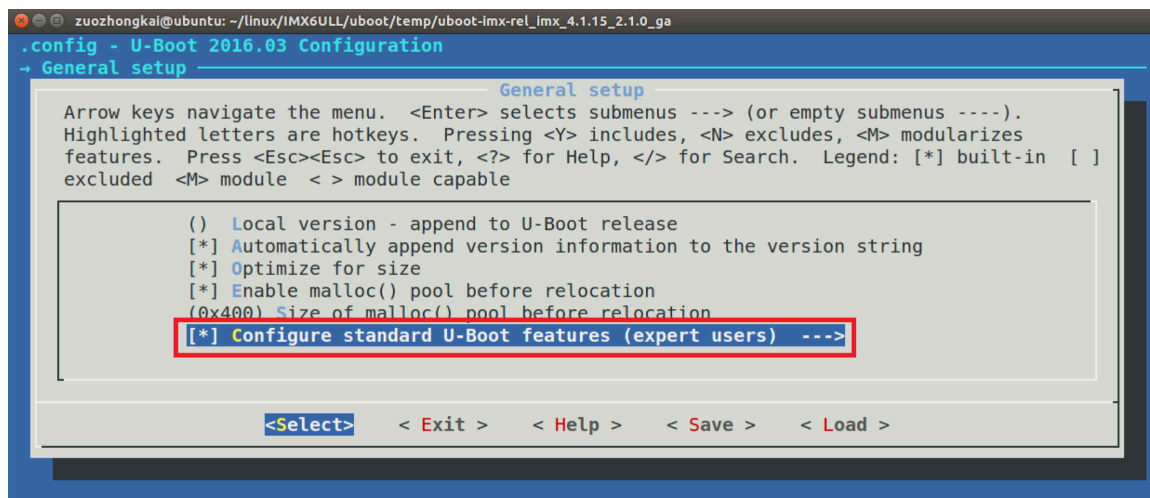


图 34.2.2.6 菜单 Configure standard U-Boot features (expert users)

从图 34.2.2.6 可以看到，前面有“[ ]”说明这个菜单是可选的，当选中这个菜单以后就可以进入到子选项中，也就是示例代码 34.2.2.9 中的第 83~99 行所描述的菜单，如图 34.2.2.7 所示：

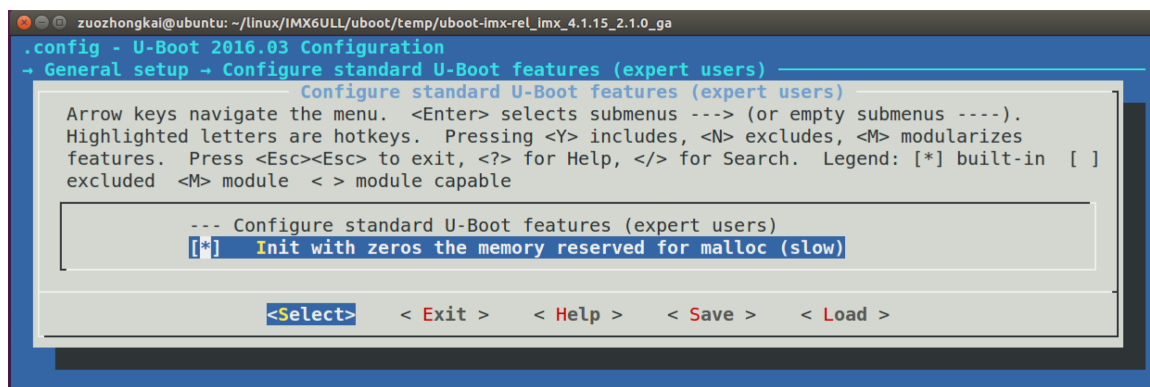


图 34.2.2.7 菜单 Init with zeros the memory reserved for malloc (slow)

如果不选择“Configure standard U-Boot features (expert users)”，那么示例代码 34.2.2.9 中的第 83~99 行所描述的菜单就不会显示出来，进去以后是空白的。

6、comment

comment 用于注释，也就是在图形化界面中显示一行注释，打开文件 drivers/mtd/nand/Kconfig，有如下所示代码：

示例代码 34.2.2.10 drivers/mtd/nand/Kconfig 代码段

```
74 config NAND_ARASAN
```

```
75      bool "Configure Arasan Nand"
76      help
......
80
81  comment "Generic NAND options"
```

第 81 行使用 comment 标注了一行注释，注释内容为：“Generic NAND options”，这行注释在配置项 NAND_ARASAN 的下面。在图形化配置界面中按照如下路径打开：

```
-> Device Drivers
   -> NAND Device Support
```

结果如图 34.2.2.8 所示：

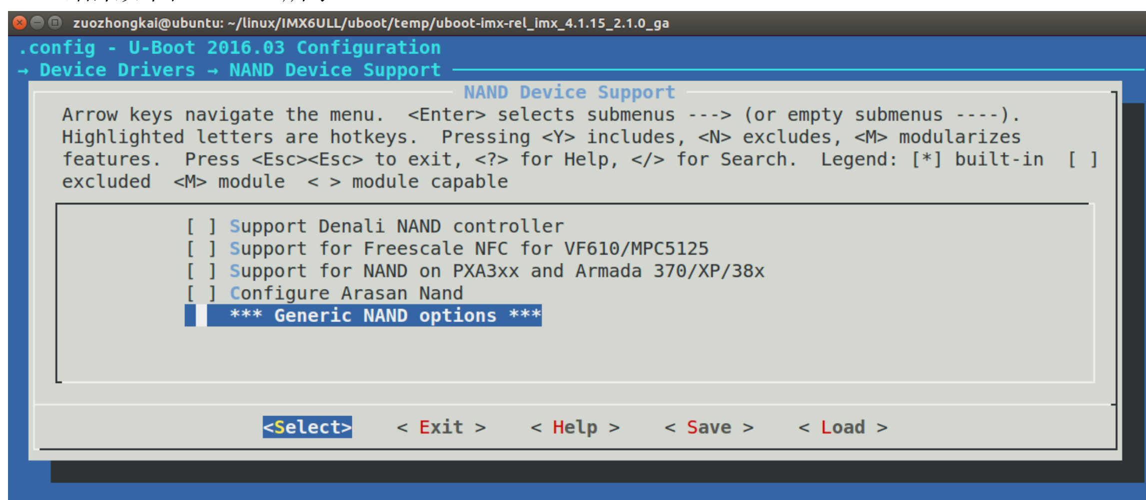


图 34.2.2.8 注释“Generic NAND options”

从图 34.2.2.8 可以看出，在配置项“Configure Arasan Nand”下面有一行注释，注释内容为“*** Generic NAND options ***”。

7、source

source 用于读取另一个 Kconfig，比如：

```
source "arch/Kconfig"
```

这个在前面已经讲过了。

Kconfig 语法就讲解到这里，基本上常用的语法就是这些，因为 uboot 相比 Linux 内核要小很多，所以配置项也要少很多，所以建议大家使用 uboot 来学习 Kconfig。一般不会修改 uboot 中的 Kconfig 文件，甚至都不会使用 uboot 的图形化界面配置工具，本小节学习 Kconfig 的目的主要还是为了 Linux 内核作准备。

34.3 添加自定义菜单

图形化配置工具的主要工作就是在.config 下面生成前缀为“CONFIG_”的变量，这些变量一般都要值，为 y，m 或 n，在 uboot 源码里面会根据这些变量来决定编译哪个文件。本小节我们就来学习一下如何添加自己的自定义菜单，自定义菜单要求如下：

①、在主界面中添加一个名为“My test menu”，此菜单内部有一个配置项。

②、配置项为“MY_TESTCONFIG”，此配置项处于菜单“My test menu”中。

③、配置项的为变量类型为 bool，默认值为 y。

④、配置项菜单名字为“This is my test config”。

⑤、配置项的帮助内容为“This is a empty config, just for tset!”。

打开顶层 Kconfig，在最后面加入如下代码：

示例代码 34.3.1 自定义菜单

```
1 menu "My test menu"
2 
3 config MY_TESTCONFIG
4   bool "This is my test config"
5   default y
6   help
7     This is a empty config, just for tset!
8 
9 endmenu       # my test menu
```

添加完成以后打开图形化配置界面，如图 34.3.1 所示：

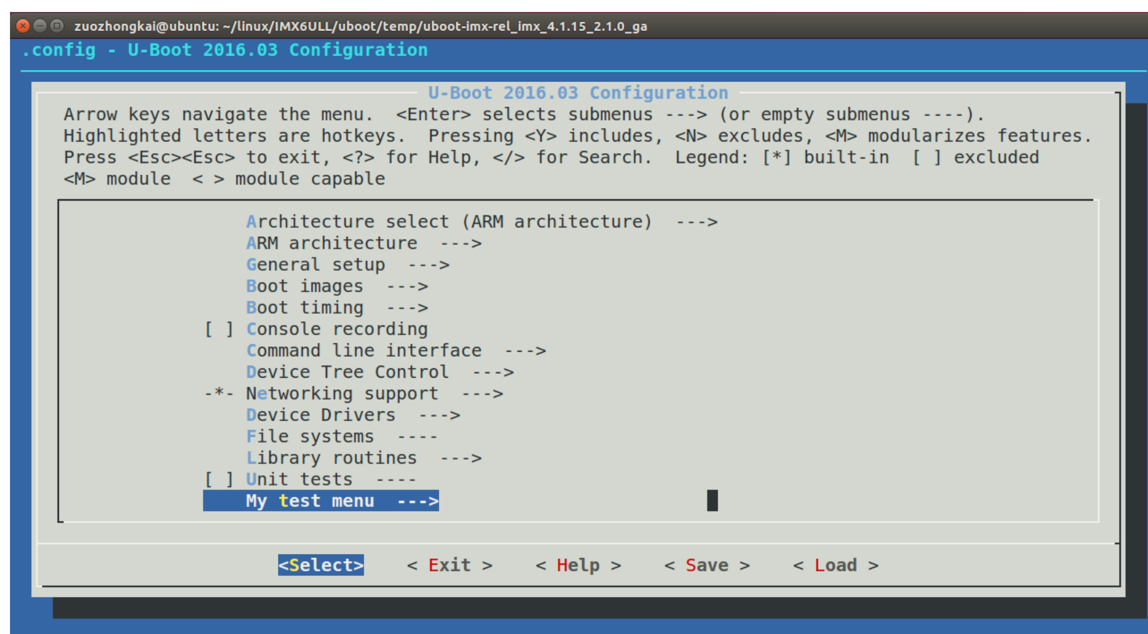


图 34.3.1 主界面

从图 34.3.1 可以看出，主菜单最后面出现了一个名为“My test menu”的子菜单，这个就是我们上面添加进来的子菜单。进入此子菜单，如图 34.3.2 所示：

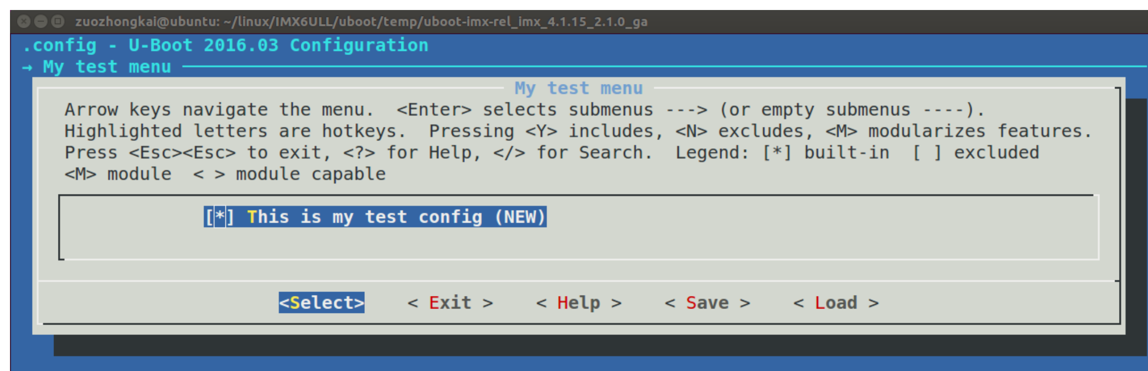


图 34.3.2 “My test menu”子菜单

从图 34.3.2 可以看出，配置项添加成功，选中“This is my test config”配置项，然后按下

“H”键打开帮助文档，如图 34.3.3 所示：

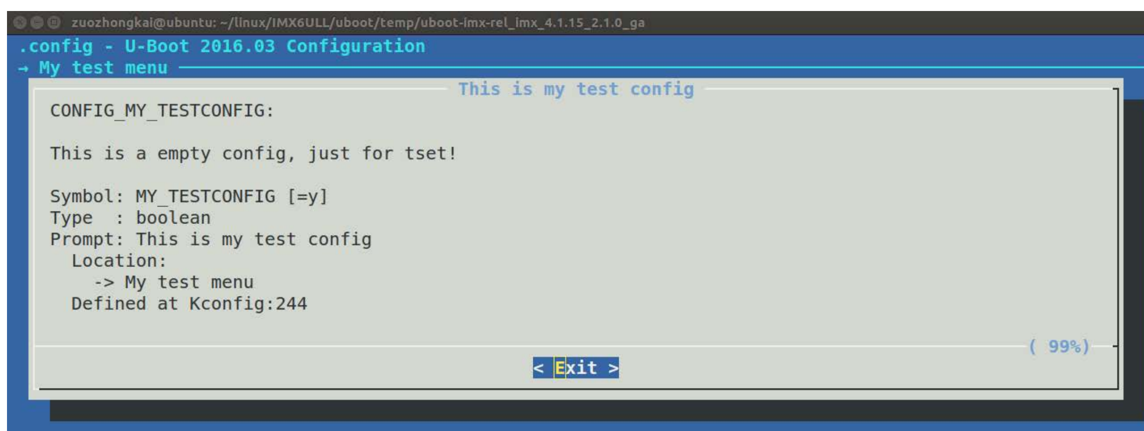


图 34.3.3 帮助信息

从图 34.3.3 可以看出，帮助信息也正确。配置项 MY_TESTCONFIG 默认也是被选中的，因此在.config 文件中肯定会有“CONFIG_MY_TESTCONFIG=y”这一行，如图 34.3.4 所示：

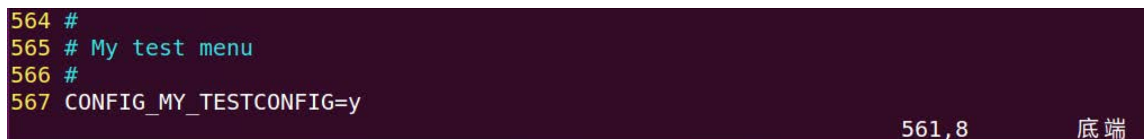


图 34.3.4 .config 文件

至此，我们在主菜单添加自己的自定义菜单就成功了，以后大家如果去半导体原厂工作的话，如果要编写 Linux 驱动，那么很有可能需要你来修改甚至编写 Kconfig 文件。Kconfig 语法其实不难，重要的点就是 34.2.2 小节中的那几个，最主要的是记住：Kconfig 文件的最终目的就是在.config 文件中生成以“CONFIG_”开头的变量。

第三十五章 Linux 内核顶层 Makefile 详解

前几章我们重点讲解了如何移植 uboot 到 I.MX6U-ALPHA 开发板上，从本章开始我们就开始学习如何移植 Linux 内核。同 uboot 一样，在具体移植之前，我们先来学习一下 Linux 内核的顶层 Makefile 文件，因为顶层 Makefile 控制着 Linux 内核的编译流程。

35.1 Linux 内核获取

关于 Linux 的起源以及发展历史，这里就不啰嗦了，网上相关的介绍太多了！即使写到这里也只是水一下教程页数而已，没有任何实际的意义。有限的时间还是放到有意义的事情上吧，Linux 由 Linux 基金会管理与发布，Linux 官网为 https://www.kernel.org，所以你想获取最新的 Linux 版本就可以在这个网站上下载，网站界面如图 35.1.1 所示：

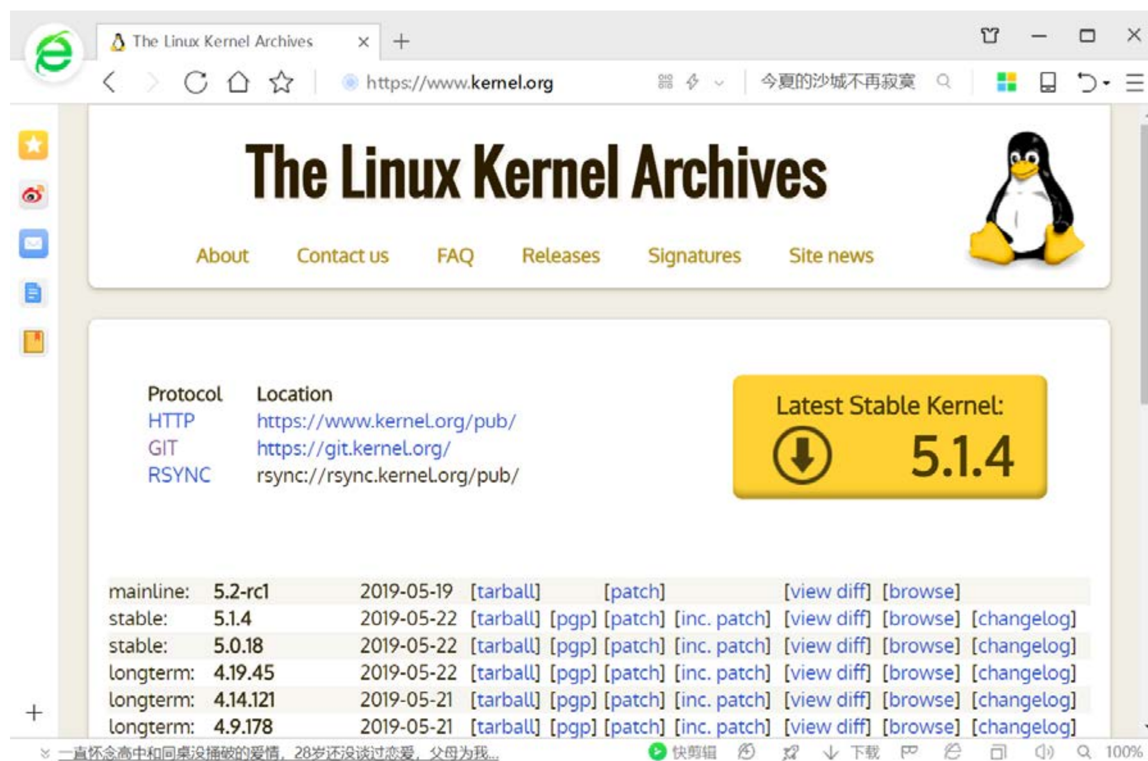


图 35.1.1 linux 官网

从图 35.1.1 可以看出最新的稳定版 Linux 已经到了 5.1.4，大家没必要追新，因为 4.x 版本的 Linux 和 5.x 版本没有本质上的区别，5.x 更多的是加入了一些新的平台、新的外设驱动而已。

NXP 会从 https://www.kernel.org 下载某个版本的 Linux 内核，然后将其移植到自己的 CPU 上，测试成功以后就会将其开放给 NXP 的 CPU 开发者。开发者下载 NXP 提供的 Linux 内核，然后将其移植到自己的产品上。本章的移植我们就使用 NXP 提供的 Linux 源码，NXP 提供 Linux 源码已经放到了开发板光盘中，路径为：1、例程源码-》4、NXP 官方原版 Uboot 和 Linux-》linux-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2。

35.2 Linux 内核编译初次编译

先看一下如何编译 Linux 源码，这里编译一下 I.MX6U-ALPHA 开发板移植好的 Linux 源码，已经放到了开发板光盘中，路径为：1、例程源码-》3、正点原子修改后的 Uboot 和 Linux-》linux-imx-4.1.15-2.1.0-g8a006db.tar.bz2。在 Ubuntu 中新建名为“alientek_linux”的文件夹，然后将 linux-imx-4.1.15-2.1.0-g8a006db.tar.bz2 这个压缩包拷贝到前面新建的 alientek_linux 文件夹中并解压，命令如下：

```
tar -vxjf linux-imx-4.1.15-2.1.0-g8a006db.tar.bz2
```

解压完成以后的 Linux 源码根目录如图 35.2.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$ ls
arch     crypto         fs       Kbuild   MAINTAINERS  README          security  virt
block    Documentation  include  Kconfig  Makefile     REPORTING-BUGS  sound
COPYING  drivers        init     kernel   mm           samples         tools
CREDITS  firmware       ipc      lib      net          scripts         usr
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$
```

图 35.2.1 正点原子提供的 Linux 源码根目录

以 EMMC 核心板为例，讲解一下如何编译出对应的 Linux 镜像文件。新建名为“mx6ull_alientek_emmc.sh”的 shell 脚本，然后在这个 shell 脚本里面输入如下所示内容：

示例代码 35.2.1 mx6ull_alientek_emmc.sh 文件内容

```
1 #!/bin/sh
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- imx_v7_defconfig
4 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
5 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- all -j16
```

使用 chmod 给予 x6ull_alientek_emmc.sh 可执行权限，然后运行此 shell 脚本，命令如下：

```
./mx6ull_alientek_emmc.sh
```

编译的时候会弹出 Linux 图形配置界面，如图 35.2.3 所示：

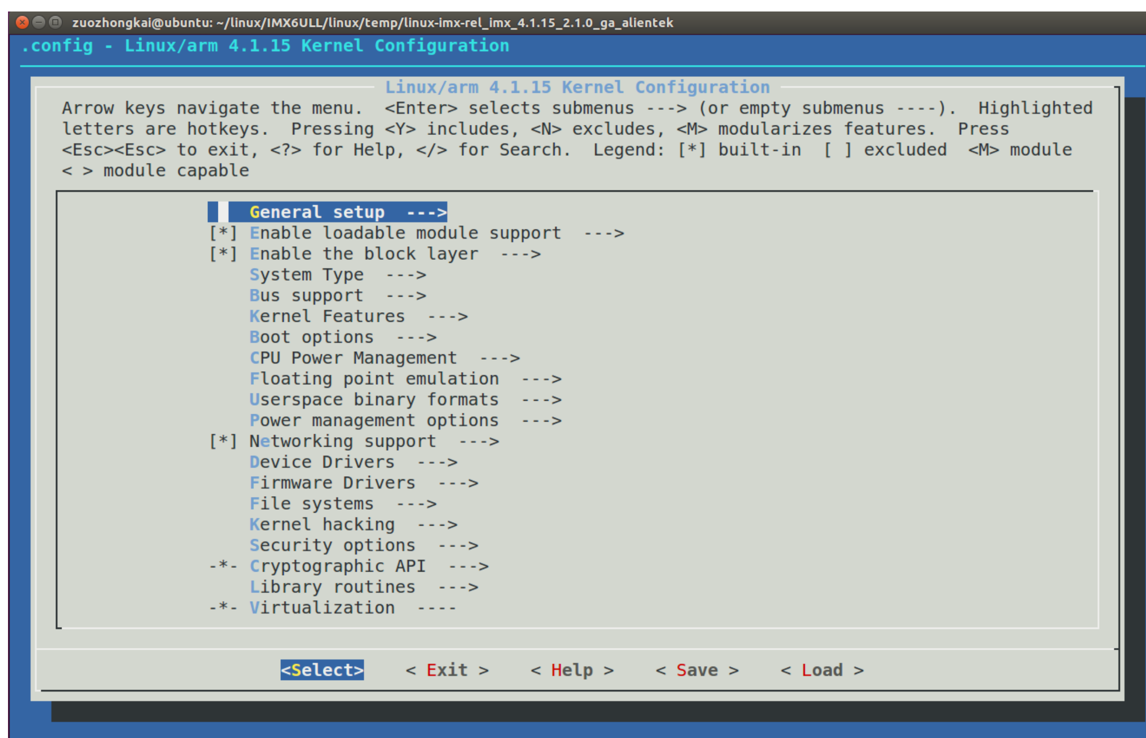


图 35.2.3 Linux 图形配置界面

Linux 的图行界面配置和 uboot 是一样的，这里我们不需要做任何的配置，直接按两下 ESC 键退出，退出图形界面以后会自动开始编译 Linux。等待编译完成，完成以后如图 35.2.4 所示：

```
zuozhongkai@ubuntu: ~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek
  LD [M]  fs/nls/nls_iso8859-15.ko
  LD [M]  lib/crc-ccitt.ko
  LD [M]  fs/udf/udf.ko
  LD [M]  fs/fat/msdos.ko
  LD [M]  lib/crc-itu-t.ko
  LD [M]  lib/crc7.ko
  LD [M]  lib/libcrc32c.ko
  LD [M]  sound/core/snd-hwdep.ko
  LD [M]  sound/core/snd-rawmidi.ko
  LD [M]  sound/usb/snd-usbmidi-lib.ko
  LD [M]  sound/usb/snd-usb-audio.ko
  AS      arch/arm/boot/compressed/piggy.lzo.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 35.2.4 Linux 编译完成

编译完成以后就会在 arch/arm/boot 这个目录下生成一个叫做 zImage 的文件，zImage 就是我们要用的 Linux 镜像文件。另外也会在 arch/arm/boo/dts 下生成很多.dtb 文件，这些.dtb 就是设备树文件。

编译 Linux 内核的时候可能会提示 "recipe for target 'arch/arm/boot/compressed/piggy.lzo' failed"，如图 35.2.5 所示：

```
/bin/sh: 1: lzop: not found
arch/arm/boot/compressed/Makefile:180: recipe for target 'arch/arm/boot/compressed/piggy.lzo' failed
make[2]: *** [arch/arm/boot/compressed/piggy.lzo] Error 1
make[2]: *** 正在等待未完成的任务....
  CC      arch/arm/boot/compressed/misc.o
arch/arm/boot/Makefile:52: recipe for target 'arch/arm/boot/compressed/vmlinux' failed
make[1]: *** [arch/arm/boot/compressed/vmlinux] Error 2
arch/arm/Makefile:316: recipe for target 'zImage' failed
```

图 35.2.5 lzop 未找到

图 35.2.5 中的错误提示 lzop 未找到，原因是没有安装 lzop 库，输入如下命令安装 lzop 库即可解决：

```
sudo apt-get install lzop
```

lzop 库安装完成以后在重新编译一下 Linux 内核即可。

看一下编译脚本 mx6ull_alientek_emmc.sh 的内容，文件内容如下：

示例代码 35.2.1 mx6ull_alientek_emmc.sh 文件内容

```
1 #!/bin/sh
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- imx_v7_defconfig
4 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
5 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- all -j16
```

第 2 行，执行 "make distclean"，清理工程，所以 mx6ull_alientek_emmc.sh 每次都会清理一下工程。如果通过图形界面配置了 Linux，但是还没保存新的配置文件，那么就要慎重使用 mx6ull_alientek_emmc.sh 编译脚本了，因为它会把你的配置信息都删除掉！

第 3 行，执行 "make xxx_defconfig"，配置工程。

第 4 行，执行 "make menuconfig"，打开图形配置界面，对 Linux 进行配置，如果不想每次编译都打开图形配置界面的话可以将这一行删除掉。

第 5 行，执行 "make"，编译 Linux 源码。

可以看出，Linux 的编译过程基本和 uboot 一样，都要先执行 "make xxx_defconfig" 来配置一下，然后在执行"make"进行编译。如果需要使用图形界面配置的话就执行"make menuconfig"。

35.3 Linux 工程目录分析

将正点原子提供的 Linux 源码进行解压，解压完成以后的目录如图 35.3.1 所示：

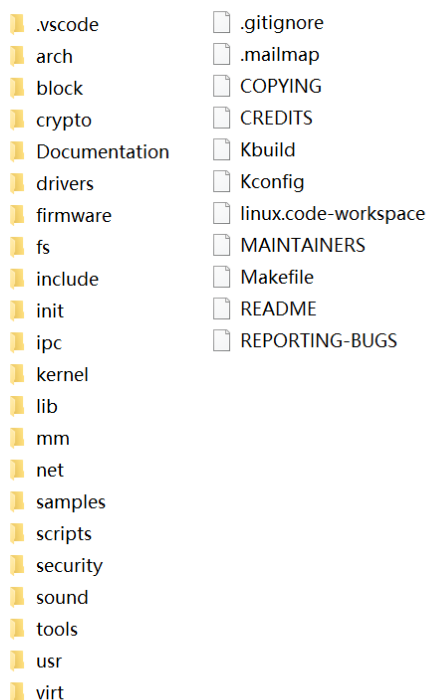


图 35.3.1 未编译的 Linux 源码目录

图 35.3.1 就是正点原子提供的未编译的 Linux 源码目录文件，我们在分析 Linux 之前一定要先在 Ubuntu 中编译一下 Linux，因为编译过程会生成一些文件，而生成的这些恰恰是分析 Linux 不可或缺的文件。编译完成以后使用 tar 压缩命令对其进行压缩并使用 Filezilla 软件将压缩后的 uboot 源码拷贝到 Windows 下。

编译后的 Linux 目录如图 35.3.2 所示：

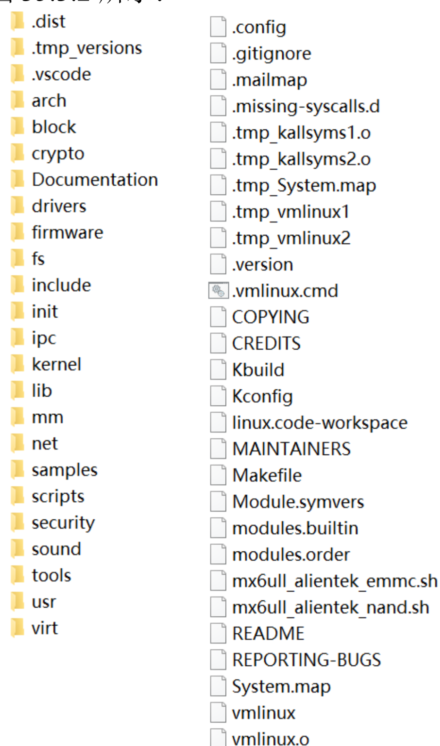


图 35.3.2 编译后的 Linux 目录

图 35.3.2 中重要的文件夹或文件的含义见表 35.3.1 所示：

<table>
<tr><th>类型</th><th>名字</th><th>描述</th><th>备注</th></tr>
<tr><td rowspan="21">文件夹</td><td>arch</td><td>架构相关目录。</td><td rowspan="20">Linux 自带</td></tr>
<tr><td>block</td><td>块设备相关目录。</td></tr>
<tr><td>crypto</td><td>加密相关目录。</td></tr>
<tr><td>Documentation</td><td>文档相关目录。</td></tr>
<tr><td>drivers</td><td>驱动相关目录。</td></tr>
<tr><td>firmeare</td><td>固件相关目录。</td></tr>
<tr><td>fs</td><td>文件系统相关目录。</td></tr>
<tr><td>include</td><td>头文件相关目录。</td></tr>
<tr><td>init</td><td>初始化相关目录。</td></tr>
<tr><td>ipc</td><td>进程间通信相关目录。</td></tr>
<tr><td>kernel</td><td>内核相关目录。</td></tr>
<tr><td>lib</td><td>库相关目录。</td></tr>
<tr><td>mm</td><td>内存管理相关目录。</td></tr>
<tr><td>net</td><td>网络相关目录。</td></tr>
<tr><td>samples</td><td>例程相关目录。</td></tr>
<tr><td>scripts</td><td>脚本相关目录。</td></tr>
<tr><td>security</td><td>安全相关目录。</td></tr>
<tr><td>sound</td><td>音频处理相关目录。</td></tr>
<tr><td>tools</td><td>工具相关目录。</td></tr>
<tr><td>usr</td><td>与 initramfs 相关的目录，用于生成 initramfs。</td></tr>
<tr><td>virt</td><td>提供虚拟机技术(KVM)。</td><td></td></tr>
<tr><td rowspan="14">文件</td><td>.config</td><td>Linux 最终使用的配置文件。</td><td>编译生成的文件。</td></tr>
<tr><td>.gitignore</td><td>git 工具相关文件。</td><td rowspan="2">Linux 自带</td></tr>
<tr><td>.mailmap</td><td>邮件列表。</td></tr>
<tr><td>.missing-syscalls.d</td><td>。</td><td>编译生成的文件</td></tr>
<tr><td>.tmp_xx</td><td>这是一系列的文件，作用目前笔者还不是很清楚。</td><td rowspan="3">编译生成的文件</td></tr>
<tr><td>.version</td><td>好像和版本有关。</td></tr>
<tr><td>.vmlinux.cmd</td><td>cmd 文件，用于连接生成 vmlinux。</td></tr>
<tr><td>COPYING</td><td>版权声明。</td><td rowspan="6">Linux 自带</td></tr>
<tr><td>CREDITS</td><td>Linux 贡献者。</td></tr>
<tr><td>Kbuild</td><td>Makefile 会读取此文件。</td></tr>
<tr><td>Kconfig</td><td>图形化配置界面的配置文件。。</td></tr>
<tr><td>MAINTAINERS</td><td>维护者名单。</td></tr>
<tr><td>Makefile</td><td>Linux 顶层 Makefile</td></tr>
<tr><td>Module.xx
modules.xx</td><td>一系列文件，和模块有关。</td><td>编译生成的文件</td></tr>
<tr><td>mx6ull_alientek_emmc.sh
mx6ull_alientek_nand.sh</td><td>正点原子提供的，Linux 编译脚本。</td><td>正点原子提供</td></tr>
</table>

	README	Linux 描述文件。	Linux 自带
	REPORTING-BUGS	BUG 上报指南	
	System.map	符号表。	编译生成的文件
	vmlinux	编译出来的、未压缩的 ELF 格式 Linux 文件	
	vmlinux.o	编译出来的 vmlinux.o 文件。	

表 35.3.1 Linux 目录

表 35.3.1 中的很多文件夹和文件我们都不需要去关心，我们要关注的文件夹或文件如下：

1、arch 目录

这个目录是和架构有关的目录，比如 arm、arm64、avr32、x86 等等架构。每种架构都对应一个目录，在这些目录中又有很多子目录，比如 boot、common、configs 等等，以 arch/arm 为例，其子目录如图 35.3.2 所示：

boot	2019/5/25 10:44	文件夹
common	2019/5/25 10:44	文件夹
configs	2019/5/25 10:44	文件夹
crypto	2019/5/25 10:44	文件夹
firmware	2019/5/25 10:44	文件夹
include	2019/5/25 10:44	文件夹
kernel	2019/5/25 10:44	文件夹
kvm	2019/5/25 10:44	文件夹
lib	2019/5/25 10:44	文件夹
mach-alpine	2019/5/25 10:44	文件夹
mach-asm9260	2019/5/25 10:44	文件夹
mach-at91	2019/5/25 10:44	文件夹
mach-axxia	2019/5/25 10:44	文件夹

图 35.3.2 arch/arm 子目录

图 35.3.2 是 arch/arm 的一部分子目录，这些子目录用于控制系统引导、系统调用、动态调频、主频设置等。arch/arm/configs 目录是不同平台的默认配置文件：xxx_defconfig，如图 35.3.3 所示：

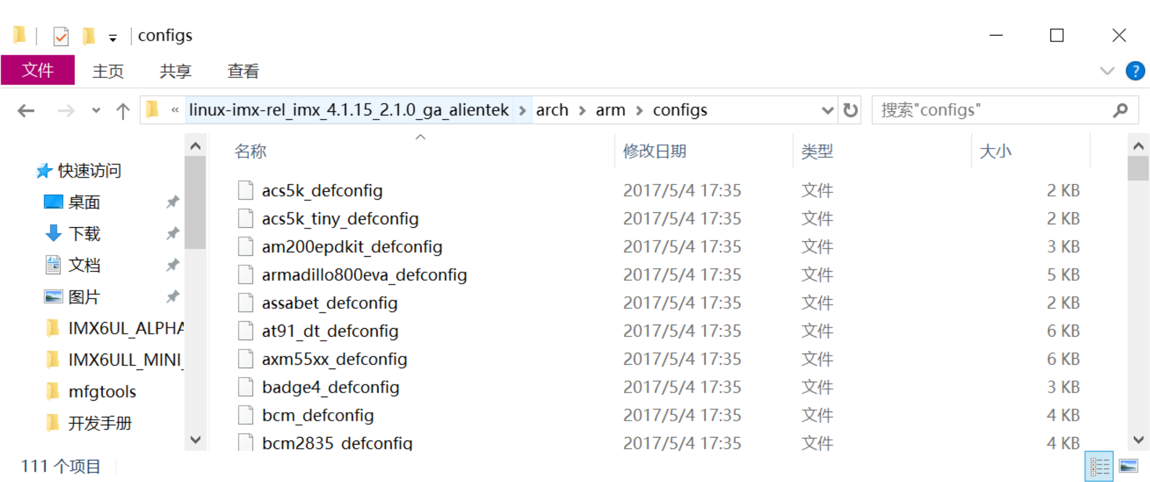


图 35.3.3 配置文件

在 arch/arm/configs 中就包含有 I.MX6U-ALPHA 开发板的默认配置文件：imx_v7_defconfig，执行“make imx_v7_defconfig”即可完成配置。arch/arm/boot/dts 目录里面是对应开发平台的设

备树文件，正点原子 I.MX6U-ALPHA 开发板对应的设备树文件如图 35.3.4 所示：

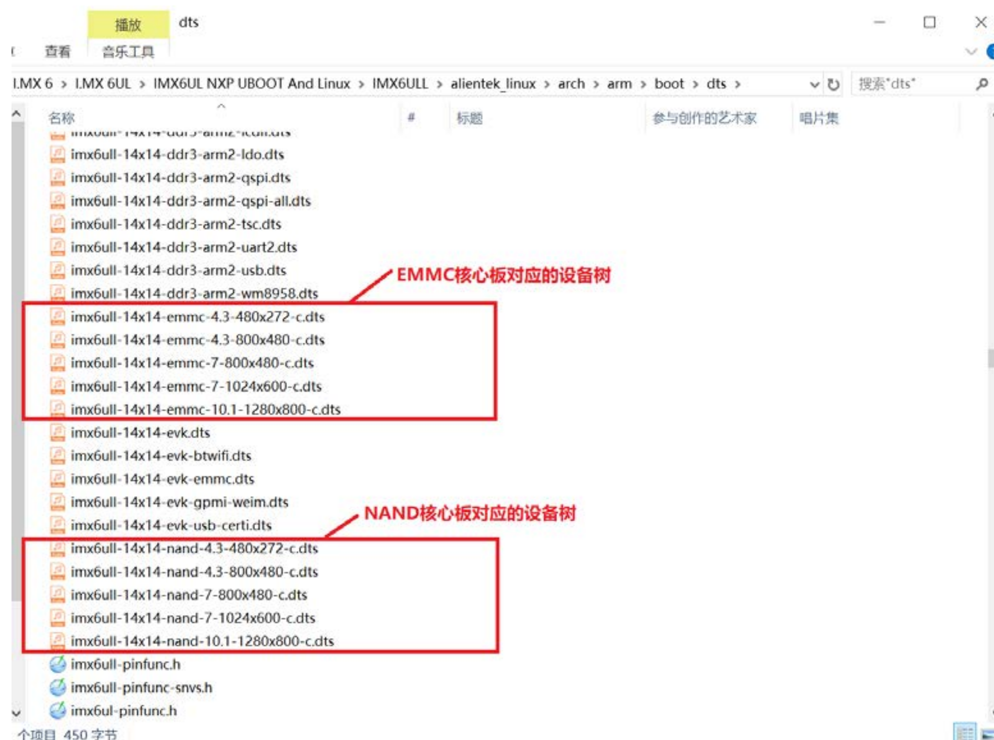


图 35.3.4 正点原子 I.MX6U 开发板对应的设备树

arch/arm/boot 目录下会保存编译出来的 Image 和 zImage 镜像文件，而 zImage 就是我们要用的 linux 镜像文件。

arch/arm/mach-xxx 目录分别为相应平台的驱动和初始化文件，比如 mach-imx 目录里面就是 I.MX 系列 CPU 的驱动和初始化文件。

2、block 目录

block 是 Linux 下块设备目录，像 SD 卡、EMMC、NAND、硬盘等存储设备就属于块设备，block 目录中存放着管理块设备的相关文件。

3、crypto 目录

crypto 目录里面存放着加密文件，比如常见的 crc、crc32、md4、md5、hash 等加密算法。

4、Documentation 目录

此目录里面存放着 Linux 相关的文档，如果要想了解 Linux 某个功能模块或驱动架构的功能，就可以在 Documentation 目录中查找有没有对应的文档。

5、drivers 目录

驱动目录文件，此目录根据驱动类型的不同，分门别类进行整理，比如 drivers/i2c 就是 I2C 相关驱动目录，drivers/gpio 就是 GPIO 相关的驱动目录，这是我们学习的重点。

6、firmware 目录

此目录用于存放固件。

7、fs 目录

此目录存放文件系统，比如 fs/ext2、fs/ext4、fs/f2fs 等，分别是 ext2、ext4 和 f2fs 等文件系统。

8、include 目录

头文件目录。

9、init 目录

此目录存放 Linux 内核启动的时候初始化代码。

10、ipc 目录

IPC 为进程间通信，ipc 目录是进程间通信的具体实现代码。

11、kernel 目录

Linux 内核代码。

12、lib 目录

lib 是库的意思，lib 目录都是一些公用的库函。

13、mm 目录

此目录存放内存管理相关代码。

14、net 目录

此目录存放网络相关代码。

15、samples 目录

此目录存放一些示例代码文件。

16、scripts 目录

脚本目录，Linux 编译的时候会用到很多脚本文件，这些脚本文件就保存在此目录中。

17、security 目录

此目录存放安全相关的文件。

18、sound 目录

此目录存放音频相关驱动文件，音频驱动文件并没有存放到 drivers 目录中，而是单独的目录。

19、tools 目录

此目录存放一些编译的时候使用到的工具。

20、usr 目录

此目录存放与 initramfs 有关的代码。

21、virt 目录

此目录存放虚拟机相关文件。

22、.config 文件

跟 uboot 一样，.config 保存着 Linux 最终的配置信息，编译 Linux 的时候会读取此文件中的配置信息。最终根据配置信息来选择编译 Linux 哪些模块，哪些功能。

23、Kbuild 文件

有些 Makefile 会读取此文件。

24、Kconfig 文件

图形化配置界面的配置文件。

25、Makefile 文件

Linux 顶层 Makefile 文件，建议好好阅读一下此文件。

26、README 文件

此文件详细讲解了如何编译 Linux 源码，以及 Linux 源码的目录信息，建议仔细阅读一下此文件。

关于 Linux 源码目录就分析到这里，接下来分析一下 Linux 的顶层 Makefile。

35.4 VSCode 工程创建

在分析 Linux 的顶层 Makefile 之前，先创建 VSCode 工程，创建过程和 uboot 一样。创建好以后将文件.vscode/settings.json 改为如下所示内容：

示例代码 35.4.1.1 settings.json 文件内容

```
1  {
2      "search.exclude": {
3          "**/node_modules": true,
4          "**/bower_components": true,
5          "**/*.o":true,
6          "**/*.su":true,
7          "**/*.cmd":true,
8          "Documentation":true,
9
10         /* 屏蔽不用的架构相关的文件 */
11         "arch/alpha":true,
12         "arch/arc":true,
13         "arch/arm64":true,
14         "arch/avr32":true,
15         "arch/[b-z]*":true,
16         "arch/arm/plat*":true,
17         "arch/arm/mach-[a-h]*":true,
18         "arch/arm/mach-[n-z]*":true,
19         "arch/arm/mach-i[n-z]*":true,
20         "arch/arm/mach-m[e-v]*":true,
21         "arch/arm/mach-k*":true,
22         "arch/arm/mach-l*":true,
23
24         /* 屏蔽排除不用的配置文件 */
25         "arch/arm/configs/[a-h]*":true,
26         "arch/arm/configs/[j-z]*":true,
27         "arch/arm/configs/imo*":true,
28         "arch/arm/configs/in*":true,
29         "arch/arm/configs/io*":true,
30         "arch/arm/configs/ix*":true,
```

```
31
32          /* 屏蔽掉不用的 DTB 文件 */
33          "arch/arm/boot/dts/[a-h]*":true,
34          "arch/arm/boot/dts/[k-z]*":true,
35          "arch/arm/boot/dts/in*":true,
36          "arch/arm/boot/dts/imx1*":true,
37          "arch/arm/boot/dts/imx7*":true,
38          "arch/arm/boot/dts/imx2*":true,
39          "arch/arm/boot/dts/imx3*":true,
40          "arch/arm/boot/dts/imx5*":true,
41          "arch/arm/boot/dts/imx6d*":true,
42          "arch/arm/boot/dts/imx6q*":true,
43          "arch/arm/boot/dts/imx6s*":true,
44          "arch/arm/boot/dts/imx6ul-*":true,
45          "arch/arm/boot/dts/imx6ull-9x9*":true,
46          "arch/arm/boot/dts/imx6ull-14x14-ddr*":true,
47      },
48      "files.exclude": {
49          "**/.git": true,
50          "**/.svn": true,
51          "**/.hg": true,
52          "**/CVS": true,
53          "**/.DS_Store": true,
54          "**/*.o":true,
55          "**/*.su":true,
56          "**/*.cmd":true,
57          "Documentation":true,
58
59          /* 屏蔽不用的架构相关的文件 */
60          "arch/alpha":true,
61          "arch/arc":true,
62          "arch/arm64":true,
63          "arch/avr32":true,
64          "arch/[b-z]*":true,
65          "arch/arm/plat*":true,
66          "arch/arm/mach-[a-h]*":true,
67          "arch/arm/mach-[n-z]*":true,
68          "arch/arm/mach-i[n-z]*":true,
69          "arch/arm/mach-m[e-v]*":true,
70          "arch/arm/mach-k*":true,
71          "arch/arm/mach-l*":true,
72
73          /* 屏蔽排除不用的配置文件 */
```

```
74          "arch/arm/configs/[a-h]*":true,
75          "arch/arm/configs/[j-z]*":true,
76          "arch/arm/configs/imo*":true,
77          "arch/arm/configs/in*":true,
78          "arch/arm/configs/io*":true,
79          "arch/arm/configs/ix*":true,
80
81          /* 屏蔽掉不用的 DTB 文件 */
82          "arch/arm/boot/dts/[a-h]*":true,
83          "arch/arm/boot/dts/[k-z]*":true,
84          "arch/arm/boot/dts/in*":true,
85          "arch/arm/boot/dts/imx1*":true,
86          "arch/arm/boot/dts/imx7*":true,
87          "arch/arm/boot/dts/imx2*":true,
88          "arch/arm/boot/dts/imx3*":true,
89          "arch/arm/boot/dts/imx5*":true,
90          "arch/arm/boot/dts/imx6d*":true,
91          "arch/arm/boot/dts/imx6q*":true,
92          "arch/arm/boot/dts/imx6s*":true,
93          "arch/arm/boot/dts/imx6ul-*":true,
94          "arch/arm/boot/dts/imx6ull-9x9*":true,
95          "arch/arm/boot/dts/imx6ull-14x14-ddr*":true,
96      }
97  }
```

创建好 VSCode 工程以后就可以开始分析 Linux 的顶层 Makefile 了。

35.5 顶层 Makefile 详解

Linux 的顶层 Makefile 和 uboot 的顶层 Makefile 非常相似，因为 uboot 参考了 Linux，前 602 行几乎一样，所以前面部分我们大致看一下就行了。

1、版本号

顶层 Makefile 一开始就是 Linux 内核的版本号，如下所示：

示例代码 35.5.1 顶层 Makefile 代码段

```
1 VERSION = 4
2 PATCHLEVEL = 1
3 SUBLEVEL = 15
4 EXTRAVERSION =
```

可以看出，Linux 内核版本号为 4.1.15。

2、MAKEFLAGS 变量

MAKEFLAGS 变量设置如下所示：

示例代码 35.5.2 顶层 Makefile 代码段

```
16 MAKEFLAGS += -rR --include-dir=$(CURDIR)
```

3、命令输出

Linux 编译的时候也可以通过“V=1”来输出完整的命令，这个和 uboot 一样，相关代码如下所示：

示例代码 35.5.3 顶层 Makefile 代码段

```
69 ifeq ("$(origin V)", "command line")
70   KBUILD_VERBOSE = $(V)
71 endif
72 ifndef KBUILD_VERBOSE
73   KBUILD_VERBOSE = 0
74 endif
75 
76 ifeq ($(KBUILD_VERBOSE),1)
77   quiet =
78   Q =
79 else
80   quiet=quiet_
81   Q = @
82 endif
```

4、静默输出

Linux 编译的时候使用“make -s”就可实现静默编译，编译的时候就不会打印任何的信息，同 uboot 一样，相关代码如下：

示例代码 35.5.4 顶层 Makefile 代码段

```
87 ifneq ($(filter 4.%,$(MAKE_VERSION)),)   # make-4
88 ifneq ($(filter %s ,$(firstword x$(MAKEFLAGS))),)
89   quiet=silent_
90 endif
91 else                    # make-3.8x
92 ifneq ($(filter s% -s%,$(MAKEFLAGS)),)
93   quiet=silent_
94 endif
95 endif
96 
97 export quiet Q KBUILD_VERBOSE
```

5、设置编译结果输出目录

Linux 编译的时候使用“O=xxx”即可将编译产生的过程文件输出到指定的目录中，相关代码如下：

示例代码 35.5.5 顶层 Makefile 代码段

```
116 ifeq ($(KBUILD_SRC),)
117 
118 # OK, Make called in directory where kernel src resides
119 # Do we want to locate output files in a separate directory?
120 ifeq ("$(origin O)", "command line")
121   KBUILD_OUTPUT := $(O)
```

```
122 endif
```

6、代码检查

Linux 也支持代码检查，使用命令“make C=1”使能代码检查，检查那些需要重新编译的文件。“make C=2”用于检查所有的源码文件，顶层 Makefile 中的代码如下：

示例代码 35.5.6 顶层 Makefile 代码段

```
172 ifeq ("$(origin C)", "command line")
173   KBUILD_CHECKSRC = $(C)
174 endif
175 ifndef KBUILD_CHECKSRC
176   KBUILD_CHECKSRC = 0
177 endif
```

7、模块编译

Linux 允许单独编译某个模块，使用命令“make M=dir”即可，旧语法“make SUBDIRS=dir”也是支持的。顶层 Makefile 中的代码如下：

示例代码 35.5.7 顶层 Makefile 代码段

```
179 # Use make M=dir to specify directory of external module to build
180 # Old syntax make ... SUBDIRS=$PWD is still supported
181 # Setting the environment variable KBUILD_EXTMOD take precedence
182 ifdef SUBDIRS
183   KBUILD_EXTMOD ?= $(SUBDIRS)
184 endif
185 
186 ifeq ("$(origin M)", "command line")
187   KBUILD_EXTMOD := $(M)
188 endif
189 
190 # If building an external module we do not care about the all: rule
191 # but instead _all depend on modules
192 PHONY += all
193 ifeq ($(KBUILD_EXTMOD),)
194 _all: all
195 else
196 _all: modules
197 endif
198 
199 ifeq ($(KBUILD_SRC),)
200         # building in the source tree
201         srctree := .
202 else
203         ifeq ($(KBUILD_SRC)/,$(dir $(CURDIR)))
204                 # building in a subdirectory of the source tree
205                 srctree := ..
```

```
206         else
207                 srctree := $(KBUILD_SRC)
208         endif
209 endif
210 objtree     := .
211 src     := $(srctree)
212 obj     := $(objtree)
213
214 VPATH       := $(srctree)$(if $(KBUILD_EXTMOD),:$(KBUILD_EXTMOD))
215
216 export srctree objtree VPATH
```

外部模块编译过程和 uboot 也一样，最终导出 srctree、objtree 和 VPATH 这三个变量的值，其中 srctree=.，也就是当前目录，objtree 同样为“.”。

8、设置目标架构和交叉编译器

同 uboot 一样，Linux 编译的时候需要设置目标板架构 ARCH 和交叉编译器 CROSS_COMPILE，在顶层 Makefile 中代码如下：

示例代码 35.5.8 顶层 Makefile 代码段

```
252 ARCH                ?= $(SUBARCH)
253 CROSS_COMPILE       ?= $(CONFIG_CROSS_COMPILE:"%"=%)
```

为了方便，一般直接修改顶层 Makefile 中的 ARCH 和 CROSS_COMPILE，直接将其设置为对应的架构和编译器，比如本教程将 ARCH 设置为为 arm，CROSS_COMPILE 设置为 arm-linux-gnueabihf-，如下所示：

示例代码 35.5.9 顶层 Makefile 代码段

```
252 ARCH                ?= arm
253 CROSS_COMPILE       ?= arm-linux-gnueabihf-
```

设置好以后我们就可以使用如下命令编译 Linux 了：

```
make xxx_defconfig          //使用默认配置文件配置 Linux
make menuconfig             //启动图形化配置界面
make -j16                   //编译 Linux
```

9、调用 scripts/Kbuild.include 文件

同 uboot 一样，Linux 顶层 Makefile 也会调用文件 scripts/Kbuild.include，顶层 Makefile 相应代码如下：

示例代码 35.5.10 顶层 Makefile 代码段

```
348 # We need some generic definitions (do not try to remake the file).
349 scripts/Kbuild.include: ;
350    include scripts/Kbuild.include
```

10、交叉编译工具变量设置

顶层 Makefile 中其他和交叉编译器有关的变量设置如下：

示例代码 35.5.11 顶层 Makefile 代码段

```
353 AS      = $(CROSS_COMPILE)as
354 LD      = $(CROSS_COMPILE)ld
355 CC      = $(CROSS_COMPILE)gcc
```

```
356 CPP     = $(CC) -E
357 AR      = $(CROSS_COMPILE)ar
358 NM      = $(CROSS_COMPILE)nm
359 STRIP       = $(CROSS_COMPILE)strip
360 OBJCOPY     = $(CROSS_COMPILE)objcopy
361 OBJDUMP     = $(CROSS_COMPILE)objdump
```

LA、LD、CC 等这些都是交叉编译器所使用的工具。

11、头文件路径变量

顶层 Makefile 定义了两个变量保存头文件路径：USERINCLUDE 和 LINUXINCLUDE，相关代码如下：

示例代码 35.5.12 顶层 Makefile 代码段

```
381 USERINCLUDE    := \
382         -I$(srctree)/arch/$(hdr-arch)/include/uapi \
383         -Iarch/$(hdr-arch)/include/generated/uapi \
384         -I$(srctree)/include/uapi \
385         -Iinclude/generated/uapi \
386                 -include $(srctree)/include/linux/kconfig.h
387 
388 # Use LINUXINCLUDE when you must reference the include/ directory.
389 # Needed to be compatible with the O= option
390 LINUXINCLUDE    := \
391         -I$(srctree)/arch/$(hdr-arch)/include \
392         -Iarch/$(hdr-arch)/include/generated/uapi \
393         -Iarch/$(hdr-arch)/include/generated \
394         $(if $(KBUILD_SRC), -I$(srctree)/include) \
395         -Iinclude \
396         $(USERINCLUDE)
```

第 381~386 行是 USERINCLUDE 是 UAPI 相关的头文件路径，第 390~396 行是 LINUXINCLUDE 是 Linux 内核源码的头文件路径。重点来看一下 LINUXINCLUDE，其中 srctree=.，hdr-arch=arm，KBUILD_SRC 为空，因此，将 USERINCLUDE 和 LINUXINCLUDE 展开以后为：

```
USERINCLUDE      := \
        -I./arch/arm/include/uapi \
        -Iarch/arm/include/generated/uapi \
        -I./include/uapi \
        -Iinclude/generated/uapi \
        -include ./include/linux/kconfig.h

LINUXINCLUDE     := \
        -I./arch/arm/include \
        -Iarch/arm/include/generated/uapi \
        -Iarch/arm/include/generated \
        -Iinclude \
```

```
            -I./arch/arm/include/uapi \
            -Iarch/arm/include/generated/uapi \
            -I./include/uapi \
            -Iinclude/generated/uapi \
            -include ./include/linux/kconfig.h
```

12、导出变量

顶层 Makefile 会导出很多变量给子 Makefile 使用，导出的这些变量如下：

示例代码 35.5.13 顶层 Makefile 代码段

```
417 export VERSION PATCHLEVEL SUBLEVEL KERNELRELEASE KERNELVERSION
418 export ARCH SRCARCH CONFIG_SHELL HOSTCC HOSTCFLAGS CROSS_COMPILE AS
LD CC
419 export CPP AR NM STRIP OBJCOPY OBJDUMP
420 export MAKE AWK GENKSYMS INSTALLKERNEL PERL PYTHON UTS_MACHINE
421 export HOSTCXX HOSTCXXFLAGS LDFLAGS_MODULE CHECK CHECKFLAGS
422
423 export KBUILD_CPPFLAGS NOSTDINC_FLAGS LINUXINCLUDE OBJCOPYFLAGS
LDFLAGS
424 export KBUILD_CFLAGS CFLAGS_KERNEL CFLAGS_MODULE CFLAGS_GCOV
CFLAGS_KASAN
425 export KBUILD_AFLAGS AFLAGS_KERNEL AFLAGS_MODULE
426 export KBUILD_AFLAGS_MODULE KBUILD_CFLAGS_MODULE
KBUILD_LDFLAGS_MODULE
427 export KBUILD_AFLAGS_KERNEL KBUILD_CFLAGS_KERNEL
428 export KBUILD_ARFLAGS
```

35.5.1 make xxx_defconfig 过程

第一次编译 Linux 之前都要使用“make xxx_defconfig”先配置 Linux 内核，在顶层 Makefile 中有“%config”这个目标，如下所示：

示例代码 35.5.1.1 顶层 Makefile 代码段

```
490 config-targets := 0
491 mixed-targets  := 0
492 dot-config     := 1
493
494 ifneq ($(filter $(no-dot-config-targets), $(MAKECMDGOALS)),)
495     ifeq ($(filter-out $(no-dot-config-targets), $(MAKECMDGOALS)),)
496         dot-config := 0
497     endif
498 endif
499
500 ifeq ($(KBUILD_EXTMOD),)
501         ifneq ($(filter config %config,$(MAKECMDGOALS)),)
502                config-targets := 1
```

```
503             ifneq ($(words $(MAKECMDGOALS)),1)
504                 mixed-targets := 1
505             endif
506         endif
507 endif
508
509 ifeq ($(mixed-targets),1)
510 # ===================================================================
511 # We're called with mixed targets (*config and build targets).
512 # Handle them one by one.
513
514 PHONY += $(MAKECMDGOALS) __build_one_by_one
515
516 $(filter-out __build_one_by_one, $(MAKECMDGOALS)):
__build_one_by_one
517     @:
518
519 __build_one_by_one:
520     $(Q)set -e; \
521     for i in $(MAKECMDGOALS); do \
522         $(MAKE) -f $(srctree)/Makefile $$i; \
523     done
524
525 else
526 ifeq ($(config-targets),1)
527 # ===================================================================
528 # *config targets only - make sure prerequisites are updated, and
529 # descend in scripts/kconfig to make the *config target
530
531 # Read arch specific Makefile to set KBUILD_DEFCONFIG as needed.
532 # KBUILD_DEFCONFIG may point out an alternative default
533 # configuration used for 'make defconfig'
534 include arch/$(SRCARCH)/Makefile
535 export KBUILD_DEFCONFIG KBUILD_KCONFIG
536
537 config: scripts_basic outputmakefile FORCE
538     $(Q)$(MAKE) $(build)=scripts/kconfig $@
539
540 %config: scripts_basic outputmakefile FORCE
541     $(Q)$(MAKE) $(build)=scripts/kconfig $@
542
543 else
......
```

```
563 endif # KBUILD_EXTMOD
```

第 490~507 行和 uboot 一样，都是设置定义变量 config-targets、mixed-targets 和 dot-config 的值，最终这三个变量的值为：

```
config-targets= 1
mixed-targets= 0
dot-config= 1
```

因为 config-targets=1，因此第 534 行~541 行成立。第 534 行引用 arch/arm/Makefile 这个文件，这个文件很重要，因为 zImage、uImage 等这些文件就是由 arch/arm/Makefile 来生成的。

第 535 行导出变量 KBUILD_DEFCONFIG KBUILD_KCONFIG。

第 537 行，没有目标与之匹配，因此不执行。

第 540 行，“make xxx_defconfig”与目标“%config”匹配，因此执行。“%config”依赖 scripts_basic、outputmakefile 和 FORCE，“%config”真正有意义的依赖就只有 scripts_basic，scripts_basic 的规则如下：

示例代码 35.5.1.2 顶层 Makefile 代码段

```
448 scripts_basic:
449     $(Q)$(MAKE) $(build)=scripts/basic
450     $(Q)rm -f .tmp_quiet_recordmcount
```

build 定义在文件 scripts/Kbuild.include 中，值为 build := -f $(srctree)/scripts/Makefile.build obj，因此将示例代码 35.5.1.2 展开就是：

```
scripts_basic:
    @make -f ./scripts/Makefile.build obj=scripts/basic    //也可以没有@，视配置而定
    @rm -f . tmp_quiet_recordmcount                        //也可以没有@
```

接着回到示例代码 35.5.1.1 的目标“%config”处，内容如下：

```
%config: scripts_basic outputmakefile FORCE
    $(Q)$(MAKE) $(build)=scripts/kconfig $@
```

将命令展开就是：

```
@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig
```

35.5.2 Makefile.build 脚本分析

从上一小节可知，“make xxx_defconfig“配置 Linux 的时候如下两行命令会执行脚本 scripts/Makefile.build：

```
@make -f ./scripts/Makefile.build obj=scripts/basic
@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig
```

我们依次来分析一下：

1、scripts_basic 目标对应的命令

scripts_basic 目标对应的命令为：@make -f ./scripts/Makefile.build obj=scripts/basic。打开文件 scripts/Makefile.build，有如下代码：

示例代码 35.5.2.1 Makefile.build 代码段

```
41 # The filename Kbuild has precedence over Makefile
42 kbuild-dir := $(if $(filter /%,$(src)),$(src),$(srctree)/$(src))
43 kbuild-file := $(if $(wildcard $(kbuild-dir)/Kbuild),$(kbuild-
dir)/Kbuild,$(kbuild-dir)/Makefile)
```

```
44 include $(kbuild-file)
```

将 kbuild-dir 展开后为：

```
kbuild-dir=./scripts/basic
```

将 kbuild-file 展开后为：

```
kbuild-file= ./scripts/basic/Makefile
```

最后将 59 行展开，即：

```
include ./scripts/basic/Makefile
```

继续分析 scripts/Makefile.build，如下代码：

示例代码 35.5.2.2 Makefile.build 代码段

```
94 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target)
$(extra-y)) \
95      $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
96      $(subdir-ym) $(always)
97     @:
```

__build 是默认目标，因为命令“@make -f ./scripts/Makefile.build obj=scripts/basic”没有指定目标，所以会使用到默认目标__build。在顶层 Makefile 中，KBUILD_BUILTIN 为 1，KBUILD_MODULES 为空，因此展开后目标__build 为：

```
__build:$(builtin-target) $(lib-target) $(extra-y)) $(subdir-ym) $(always)
    @:
```

可以看出目标__build 有 5 个依赖：builtin-target、lib-target、extra-y、subdir-ym 和 always。这 5 个依赖的具体内容如下：

```
builtin-target =
lib-target =
extra-y =
subdir-ym =
always = scripts/basic/fixdep scripts/basic/bin2c
```

只有 always 有效，因此__build 最终为：

```
__build: scripts/basic/fixdep scripts/basic/bin2c
    @:
```

__build 依赖于 scripts/basic/fixdep 和 scripts/basic/bin2c，所以要先将 scripts/basic/fixdep 和 scripts/basic/bin2c.c 这两个文件编译成 fixdep 和 bin2c。

综上所述，scripts_basic 目标的作用就是编译出 scripts/basic/fixdep 和 scripts/basic/bin2c 这两个软件。

2、 %config 目标对应的命令

%config 目标对应的命令为：@make -f ./scripts/Makefile.build obj=scripts/kconfig xxx_defconfig，此命令会使用到的各个变量值如下：

```
src= scripts/kconfig
kbuild-dir = ./scripts/kconfig
kbuild-file = ./scripts/kconfig/Makefile
include ./scripts/kconfig/Makefile
```

可以看出，Makefile.build 会读取 scripts/kconfig/Makefile 中的内容，此文件有如下所示内容：

示例代码 35.5.2.3 scripts/kconfig/Makefile 代码段

```
113 %_defconfig: $(obj)/conf
114     $(Q)$< $(silent) --defconfig=arch/$(SRCARCH)/configs/$@
$(Kconfig)
```

目标%_defconfig 与 xxx_defconfig 匹配，所以会执行这条规则，将其展开就是：

```
%_defconfig: scripts/kconfig/conf
    @ scripts/kconfig/conf   --defconfig=arch/arm/configs/%_defconfig   Kconfig
```

%_defconfig 依赖 scripts/kconfig/conf，所以会编译 scripts/kconfig/conf.c 生成 conf 这个软件。此软件就会将%_defconfig 中的配置输出到.config 文件中，最终生成 Linux kernel 根目录下的.config 文件。

35.5.3 make 过程

使用命令“make xxx_defconfig”配置好 Linux 内核以后就可以使用“make”或者“make all”命令进行编译。顶层 Makefile 有如下代码：

示例代码 35.5.3.1 顶层 Makefile 代码段

```
125 PHONY := _all
126 _all:
......
192 PHONY += all
193 ifeq ($(KBUILD_EXTMOD),)
194 _all: all
195 else
196 _all: modules
197 endif
......
608 all: vmlinux
```

第 126 行，_all 是默认目标，如果使用命令“make”编译 Linux 的话此目标就会被匹配。

第 193 行，如果 KBUILD_EXTMOD 为空的话 194 行的代码成立。

第 194 行，默认目标_all 依赖 all。

第 608 行，目标 all 依赖 vmlinux，所以接下来的重点就是 vmlinux！

顶层 Makefile 中有如下代码：

示例代码 35.5.3.2 顶层 Makefile 代码段

```
904 # Externally visible symbols (used by link-vmlinux.sh)
905 export KBUILD_VMLINUX_INIT := $(head-y) $(init-y)
906 export KBUILD_VMLINUX_MAIN := $(core-y) $(libs-y) $(drivers-y)
$(net-y)
907 export KBUILD_LDS          := arch/$(SRCARCH)/kernel/vmlinux.lds
908 export LDFLAGS_vmlinux
909 # used by scripts/pacmage/Makefile
910 export KBUILD_ALLDIRS := $(sort $(filter-out arch/%,$(vmlinux-
alldirs)) arch Documentation include samples scripts tools virt)
911
912 vmlinux-deps := $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT)
$(KBUILD_VMLINUX_MAIN)
```

```
913
914 # Final link of vmlinux
915      cmd_link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS)
$(LDFLAGS_vmlinux)
916 quiet_cmd_link-vmlinux = LINK    $@
917
918 # Include targets which we want to
919 # execute if the rest of the kernel build went well.
920 vmlinux: scripts/link-vmlinux.sh $(vmlinux-deps) FORCE
921 ifdef CONFIG_HEADERS_CHECK
922    $(Q)$(MAKE) -f $(srctree)/Makefile headers_check
923 endif
924 ifdef CONFIG_SAMPLES
925    $(Q)$(MAKE) $(build)=samples
926 endif
927 ifdef CONFIG_BUILD_DOCSRC
928    $(Q)$(MAKE) $(build)=Documentation
929 endif
930 ifdef CONFIG_GDB_SCRIPTS
931    $(Q)ln -fsn `cd $(srctree) && /bin/pwd`/scripts/gdb/vmlinux-
gdb.py
932 endif
933    +$(call if_changed,link-vmlinux)
```

从第 920 行可以看出目标 vmlinux 依赖 scripts/link-vmlinux.sh $(vmlinux-deps) FORCE。第 912 行定义了 vmlinux-deps，值为：

```
vmlinux-deps= $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)
```

第 905 行，KBUILD_VMLINUX_INIT= $(head-y) $(init-y)。

第 906 行，KBUILD_VMLINUX_MAIN = $(core-y) $(libs-y) $(drivers-y) $(net-y)。

第 907 行，KBUILD_LDS= arch/$(SRCARCH)/kernel/vmlinux.lds，其中 SRCARCH=arm，因此 KBUILD_LDS= arch/arm/kernel/vmlinux.lds。

综上所述，vmlinux 的依赖为：scripts/link-vmlinux.sh、$(head-y) 、$(init-y)、$(core-y) 、$(libs-y) 、$(drivers-y) 、$(net-y)、arch/arm/kernel/vmlinux.lds 和 FORCE。

第 933 行的命令用于链接生成 vmlinux。

重点来看一下$(head-y) 、$(init-y)、$(core-y) 、$(libs-y) 、$(drivers-y) 和$(net-y)这六个变量的值。

1、head-y

head-y 定义在文件 arch/arm/Makefile 中，内容如下：

示例代码 35.5.3.3 arch/arm/Makefile 代码段

```
135 head-y      := arch/arm/kernel/head$(MMUEXT).o
```

当不使能 MMU 的话 MMUEXT=-nommu，如果使能 MMU 的话为空，因此 head-y 最终的值为：

```
head-y = arch/arm/kernel/head.o
```

2、init-y、drivers-y 和 net-y

在顶层 Makefile 中有如下代码：

示例代码 35.5.3.4 顶层 Makefile 代码段

```
558 init-y          := init/
559 drivers-y       := drivers/ sound/ firmware/
560 net-y           := net/
......
896 init-y          := $(patsubst %/, %/built-in.o, $(init-y))
898 drivers-y       := $(patsubst %/, %/built-in.o, $(drivers-y))
899 net-y           := $(patsubst %/, %/built-in.o, $(net-y))
```

从示例代码 35.5.3.4 可知，init-y、libs-y、drivers-y 和 net-y 最终的值为：

```
init-y      = init/built-in.o
drivers-y = drivers/built-in.o    sound/built-in.o    firmware/built-in.o
net-y       = net/built-in.o
```

3、libs-y

libs-y 基本和 init-y 一样，在顶层 Makefile 中存在如下代码：

示例代码 35.5.3.5 顶层 Makefile 代码段

```
561 libs-y      := lib/
......
900 libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
901 libs-y2     := $(patsubst %/, %/built-in.o, $(libs-y))
902 libs-y      := $(libs-y1) $(libs-y2)
```

根据示例代码 35.5.3.5 可知，libs-y 应该等于“lib.a built-in.o”，这个只正确了一部分！因为在 arch/arm/Makefile 中会向 libs-y 中追加一些值，代码如下：

示例代码 35.5.3.6 arch/arm/Makefile 代码段

```
286 libs-y     := arch/arm/lib/ $(libs-y)
```

arch/arm/Makefile 将 libs-y 的值改为了：arch/arm/lib $(libs-y)，展开以后为：

```
libs-y = arch/arm/lib lib/
```

因此根据示例代码 35.5.3.5 的第 900~902 行可知，libs-y 最终应该为：

```
libs-y = arch/arm/lib/lib.a    lib/lib.a    arch/arm/lib/built-in.o    lib/built-in.o
```

4、core-y

core-y 和 init-y 也一样，在顶层 Makefile 中有如下代码：

示例代码 35.5.3.7 顶层 Makefile 代码段

```
532 core-y     := usr/
......
887 core-y     += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

但是在 arch/arm/Makefile 中会对 core-y 进行追加，代码如下：

示例代码 35.5.3.8 arch/arm/Makefile 代码段

```
269 core-$(CONFIG_FPE_NWFPE)    += arch/arm/nwfpe/
270 core-$(CONFIG_FPE_FASTFPE)  += $(FASTFPE_OBJ)
271 core-$(CONFIG_VFP)          += arch/arm/vfp/
272 core-$(CONFIG_XEN)          += arch/arm/xen/
```

```
273 core-$(CONFIG_KVM_ARM_HOST) += arch/arm/kvm/
274 core-$(CONFIG_VDSO)         += arch/arm/vdso/
275
276 # If we have a machine-specific directory, then include it in the
build.
277 core-y              += arch/arm/kernel/ arch/arm/mm/ arch/arm/common/
278 core-y              += arch/arm/probes/
279 core-y              += arch/arm/net/
280 core-y              += arch/arm/crypto/
281 core-y              += arch/arm/firmware/
282 core-y              += $(machdirs) $(platdirs)
```

第 269~274 行根据不同的配置向 core-y 追加不同的值，比如使能 VFP 的话就会在.config 中有 CONFIG_VFP=y 这一行，那么 core-y 就会追加“arch/arm/vfp/”。

第 277~282 行就是对 core-y 直接追加的值。

在顶层 Makefile 中有如下一行：

示例代码 35.5.3.9 顶层 Makefile 代码段

```
897 core-y     := $(patsubst %/, %/built-in.o, $(core-y))
```

经过上述代码的转换，最终 core-y 的值为：

```
core-y = usr/built-in.o                 arch/arm/vfp/built-in.o \
         arch/arm/vdso/built-in.o       arch/arm/kernel/built-in.o \
         arch/arm/mm/built-in.o         arch/arm/common/built-in.o \
         arch/arm/probes/built-in.o     arch/arm/net/built-in.o \
         arch/arm/crypto/built-in.o     arch/arm/firmware/built-in.o \
         arch/arm/mach-imx/built-in.o   kernel/built-in.o\
         mm/built-in.o                  fs/built-in.o \
         ipc/built-in.o                 security/built-in.o \
         crypto/built-in.o              block/built-in.o
```

关于 head-y 、init-y、core-y 、libs-y 、drivers-y 和 net-y 这 6 个变量就讲解到这里。这些变量都是一些 built-in.o 或.a 等文件，这个和 uboot 一样，都是将相应目录中的源码文件进行编译，然后在各自目录下生成 built-in.o 文件，有些生成了.a 库文件。最终将这些 built-in.o 和.a 文件进行链接即可形成 ELF 格式的可执行文件，也就是 vmlinux！但是链接是需要连接脚本的，vmlinux 的依赖 arch/arm/kernel/vmlinux.lds 就是整个 Linux 的链接脚本。

示例代码 35.5.3.2 第 933 行的命令“+$(call if_changed,link-vmlinux)”表示将$(call if_changed,link-vmlinux)的结果作为最终生成 vmlinux 的命令，前面的“+”表示该命令结果不可忽略。$(call if_changed,link-vmlinux)是调用函数 if_changed，link-vmlinux 是函数 if_changed 的参数，函数 if_changed 定义在文件 scripts/Kbuild.include 中，如下所示：

示例代码 35.5.3.10 scripts/Kbuild.include 代码段

```
247 if_changed = $(if $(strip $(any-prereq) $(arg-check)),          \
248     @set -e;                                                     \
249     $(echo-cmd) $(cmd_$(1));                                     \
250     printf '%s\n' 'cmd_$@ := $(make-cmd)' > $(dot-target).cmd)
```

any-prereq 用于检查依赖文件是否有变化，如果依赖文件有变化那么 any-prereq 就不为空，否则就为空。arg-check 用于检查参数是否有变化，如果没有变化那么 arg-check 就为空。

第 248 行，“@set -e”告诉 bash，如果任何语句的执行结果不为 true(也就是执行出错)的话就直接退出。

第 249 行，$(echo-cmd)用于打印命令执行过程，比如在链接 vmlinux 的时候就会输出“LINK vmlinux”。$(cmd_$(1))中的$(1)表示参数，也就是 link-vmlinux，因此$(cmd_$(1))表示执行 cmd_link-vmlinux 的内容。cmd_link-vmlinux 在顶层 Makefile 中有如下所示定义：

示例代码 35.5.3.11 顶层 Makefile 代码段

```
914 # Final link of vmlinux
915       cmd_link-vmlinux = $(CONFIG_SHELL) $< $(LD) $(LDFLAGS)
$(LDFLAGS_vmlinux)
916 quiet_cmd_link-vmlinux = LINK    $@
```

第 915 行就是 cmd_link-vmlinux 的值，其中 CONFIG_SHELL=/bin/bash，$<表示目标 vmlinux 的第一个依赖文件，根据示例代码 35.5.3.2 可知，这个文件为 scripts/link-vmlinux.sh。LD= arm-linux-gnueabihf-ld -EL，LDFLAGS 为空。LDFLAGS_vmlinux 的值由顶层 Makefile 和 arch/arm/Makefile 这两个文件共同决定，最终 LDFLAGS_vmlinux=-p --no-undefined -X --pic-veneer --build-id。因此 cmd_link-vmlinux 最终的值为：

cmd_link-vmlinux = /bin/bash scripts/link-vmlinux.sh arm-linux-gnueabihf-ld -EL -p --no-undefined -X --pic-veneer --build-id

cmd_link-vmlinux 会调用 scripts/link-vmlinux.sh 这个脚本来链接出 vmlinux！在 link-vmlinux.sh 中有如下所示代码：

示例代码 35.5.3.12 scripts/link-vmlinux.sh 代码段

```
51 vmlinux_link()
52 {
53  local lds="${objtree}/${KBUILD_LDS}"
54
55  if [ "${SRCARCH}" != "um" ]; then
56      ${LD} ${LDFLAGS} ${LDFLAGS_vmlinux} -o ${2}                  \
57          -T ${lds} ${KBUILD_VMLINUX_INIT}                    \
58          --start-group ${KBUILD_VMLINUX_MAIN} --end-group ${1}
59  else
60      ${CC} ${CFLAGS_vmlinux} -o ${2}                          \
61          -Wl,-T,${lds} ${KBUILD_VMLINUX_INIT}                 \
62          -Wl,--start-group                                 \
63              ${KBUILD_VMLINUX_MAIN}                        \
64          -Wl,--end-group                                   \
65          -lutil ${1}
66      rm -f linux
67  fi
68 }
......
216 info LD vmlinux
217 vmlinux_link "${kallsymso}" vmlinux
```

vmliux_link 就是最终链接出 vmlinux 的函数，第 55 行判断 SRCARCH 是否等于“um”，如果不相等的话就执行 56~58 行的代码。因为 SRCARCH=arm，因此条件成立，执行 56~58 行的

代码。这三行代码就应该很熟悉了！就是普通的链接操作，连接脚本为 lds= ./arch/arm/kernel/vmlinux.lds，需要链接的文件由变量 KBUILD_VMLINUX_INIT 和 KBUILD_VMLINUX_MAIN 来决定，这两个变量在示例代码 35.5.3.2 中已经讲解过了。

第 217 行调用 vmlinux_link 函数来链接出 vmlinux。

使用命令"make V=1"编译 Linux，会有如图 35.5.3.1 所示的编译信息：

```
+ arm-linux-gnueabihf-ld -EL -p --no-undefined -X --pic-veneer --build-id -o vmlinux -T ./arch/arm/kernel/v
mlinux.lds arch/arm/kernel/head.o init/built-in.o --start-group usr/built-in.o arch/arm/vfp/built-in.o arch
/arm/vdso/built-in.o arch/arm/kernel/built-in.o arch/arm/mm/built-in.o arch/arm/common/built-in.o arch/arm/
probes/built-in.o arch/arm/net/built-in.o arch/arm/crypto/built-in.o arch/arm/firmware/built-in.o arch/arm/
mach-imx/built-in.o kernel/built-in.o mm/built-in.o fs/built-in.o ipc/built-in.o security/built-in.o crypto
/built-in.o block/built-in.o arch/arm/lib/lib.a lib/lib.a arch/arm/lib/built-in.o lib/built-in.o drivers/bu
ilt-in.o sound/built-in.o firmware/built-in.o net/built-in.o --end-group .tmp_kallsyms2.o
```

图 35.5.3.1 link-vmlinux.sh 链接 vmlinux 过程

至此我们基本理清了 make 的过程，重点就是将各个子目录下的 built-in.o、.a 等文件链接在一起，最终生成 vmlinux 这个 ELF 格式的可执行文件。链接脚本为 arch/arm/kernel/vmlinux.lds，链接过程是由 shell 脚本 scripts/link-vmlinux.s 来完成的。接下来的问题就是这些子目录下的 built-in.o、.a 等文件又是如何编译出来的呢？

35.5.4 built-in.o 文件编译生成过程

根据示例代码 35.5.3.2 第 920 行可知，vmliux 依赖 vmlinux-deps，而 vmlinux-deps= $(KBUILD_LDS) $(KBUILD_VMLINUX_INIT) $(KBUILD_VMLINUX_MAIN)，KBUILD_LDS 是连接脚本，这里不考虑，剩下的 KBUILD_VMLINUX_INIT 和 KBUILD_VMLINUX_MAIN 就是各个子目录下的 built-in.o、.a 等文件。最终 vmlinux-deps 的值如下：

```
vmlinux-deps =  arch/arm/kernel/vmlinux.lds      arch/arm/kernel/head.o \
                init/built-in.o                  usr/built-in.o \
                arch/arm/vfp/built-in.o          arch/arm/vdso/built-in.o \
                arch/arm/kernel/built-in.o       arch/arm/mm/built-in.o \
                arch/arm/common/built-in.o       arch/arm/probes/built-in.o \
                arch/arm/net/built-in.o          arch/arm/crypto/built-in.o \
                arch/arm/firmware/built-in.o     arch/arm/mach-imx/built-in.o \
                kernel/built-in.o                mm/built-in.o \
                fs/built-in.o                    ipc/built-in.o \
                security/built-in.o              crypto/built-in.o\
                block/built-in.o                 arch/arm/lib/lib.a\
                lib/lib.a                        arch/arm/lib/built-in.o\
                lib/built-in.o                   drivers/built-in.o \
                sound/built-in.o                 firmware/built-in.o \
                net/built-in.o
```

除了 arch/arm/kernel/vmlinux.lds 以外，其他都是要编译链接生成的。在顶层 Makefile 中有如下代码：

示例代码 35.5.4.1 顶层 Makefile 代码段

```
937 $(sort $(vmlinux-deps)): $(vmlinux-dirs) ;
```

sort 是排序函数，用于对 vmlinux-deps 的字符串列表进行排序，并且去掉重复的单词。可以看出 vmlinux-deps 依赖 vmlinux-dirs，vmlinux-dirs 也定义在顶层 Makefile 中，定义如下：

示例代码 35.5.4.2 顶层 Makefile 代码段

```
889 vmlinux-dirs   := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
```

```
890             $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
891             $(net-y) $(net-m) $(libs-y) $(libs-m)))
```

vmlinux-dirs 看名字就知道和目录有关，此变量保存着生成 vmlinux 所需源码文件的目录，值如下：

```
vmlinux-dirs = init             usr                 arch/arm/vfp \
               arch/arm/vdso    arch/arm/kernel     arch/arm/mm \
               arch/arm/common  arch/arm/probes     arch/arm/net \
               arch/arm/crypto  arch/arm/firmware   arch/arm/mach-imx\
               kernel           mm                  fs \
               ipc              security            crypto \
               block            drivers             sound \
               firmware         net                 arch/arm/lib \
               lib
```

在顶层 Makefile 中有如下代码：

示例代码 35.5.4.3 顶层 Makefile 代码段

```
946 $(vmlinux-dirs): prepare scripts
947    $(Q)$(MAKE) $(build)=$@
```

目标 vmlinux-dirs 依赖 prepare 和 scripts，这两个依赖不去浪费时间了，重点看一下第 947 行的命令。build 前面已经说了，值为“-f ./scripts/Makefile.build obj”，因此将 947 行的命令展开就是：

```
@ make -f ./scripts/Makefile.build obj=$@
```

$@表示目标文件，也就是 vmlinux-dirs 的值，将 vmlinux-dirs 中的这些目录全部带入到命令中，结果如下：

```
@ make -f ./scripts/Makefile.build obj=init
@ make -f ./scripts/Makefile.build obj=usr
@ make -f ./scripts/Makefile.build obj=arch/arm/vfp
@ make -f ./scripts/Makefile.build obj=arch/arm/vdso
@ make -f ./scripts/Makefile.build obj=arch/arm/kernel
@ make -f ./scripts/Makefile.build obj=arch/arm/mm
@ make -f ./scripts/Makefile.build obj=arch/arm/common
@ make -f ./scripts/Makefile.build obj=arch/arm/probes
@ make -f ./scripts/Makefile.build obj=arch/arm/net
@ make -f ./scripts/Makefile.build obj=arch/arm/crypto
@ make -f ./scripts/Makefile.build obj=arch/arm/firmware
@ make -f ./scripts/Makefile.build obj=arch/arm/mach-imx
@ make -f ./scripts/Makefile.build obj=kernel
@ make -f ./scripts/Makefile.build obj=mm
@ make -f ./scripts/Makefile.build obj=fs
@ make -f ./scripts/Makefile.build obj=ipc
@ make -f ./scripts/Makefile.build obj=security
@ make -f ./scripts/Makefile.build obj=crypto
@ make -f ./scripts/Makefile.build obj=block
@ make -f ./scripts/Makefile.build obj=drivers
```

```
@ make -f ./scripts/Makefile.build obj=sound
@ make -f ./scripts/Makefile.build obj=firmware
@ make -f ./scripts/Makefile.build obj=net
@ make -f ./scripts/Makefile.build obj=arch/arm/lib
@ make -f ./scripts/Makefile.build obj=lib
```

这些命令运行过程其实都是一样的，我们就以“@ make -f ./scripts/Makefile.build obj=init”这个命令为例，讲解一下详细的运行过程。这里又要用到 Makefile.build 这个脚本了，此脚本默认目标为__build，这个在 35.5.2 小节已经讲过了，我们再来看一下，__build 目标对应的规则如下：

示例代码 35.5.4.4 scripts/Makefile.build 代码段

```
94 __build: $(if $(KBUILD_BUILTIN),$(builtin-target) $(lib-target)
$(extra-y)) \
95   $(if $(KBUILD_MODULES),$(obj-m) $(modorder-target)) \
96   $(subdir-ym) $(always)
97  @:
```

当只编译 Linux 内核镜像文件，也就是使用“make zImage”编译的时候，KBUILD_BUILTIN=1，KBUILD_MODULES 为空。“make”命令是会编译所有的东西，包括 Linux 内核镜像文件和一些模块文件。如果只编译 Linux 内核镜像的话，__build 目标简化为：

```
__build: $(builtin-target) $(lib-target) $(extra-y)) $(subdir-ym) $(always)
 @:
```

重点来看一下 builtin-target 这个依赖，builtin-target 同样定义在文件 scripts/Makefile.build 中，定义如下：

示例代码 35.5.4.5 scripts/Makefile.build 代码段

```
86 ifneq ($(strip $(obj-y) $(obj-m) $(obj-) $(subdir-m) $(lib-
target)),)
87 builtin-target := $(obj)/built-in.o
88 endif
```

第 87 行就是 builtin-target 变量的值，为“$(obj)/built-in.o”，这就是这些 built-in.o 的来源了。要生成 built-in.o，要求 obj-y、obj-m、obj-、subdir-m 和 lib-target 这些变量不能全部为空。最后一个问题：built-in.o 是怎么生成的？在文件 scripts/Makefile.build 中有如下代码：

示例代码 35.5.4.6 顶层 Makefile 代码段

```
325 #
326 # Rule to compile a set of .o files into one .o file
327 #
328 ifdef builtin-target
329 quiet_cmd_link_o_target = LD      $@
330 # If the list of objects to link is empty, just create an empty
built-in.o
331 cmd_link_o_target = $(if $(strip $(obj-y)),\
332             $(LD) $(ld_flags) -r -o $@ $(filter $(obj-y), $^) \
333             $(cmd_secanalysis),\
334             rm -f $@; $(AR) rcs$(KBUILD_ARFLAGS) $@)
335
```

```
336 $(builtin-target): $(obj-y) FORCE
337     $(call if_changed,link_o_target)
338
339 targets += $(builtin-target)
340 endif # builtin-target
```

第 336 行的目标就是 builtin-target，依赖为 obj-y，命令为“$(call if_changed,link_o_target)”，也就是调用函数 if_changed，参数为 link_o_target，其返回值就是具体的命令。前面讲过了 if_changed，它会调用 cmd_$(1)所对应的命令($(1)就是函数的第 1 个参数)，在在这里就是调用 cmd_link_o_target 所对应的命令，也就是第 331~334 行的命令。cmd_link_o_target 就是使用 LD 将某个目录下的所有.o 文件链接在一起，最终形成 built-in.o。

35.5.5 make zImage 过程

1、vmlinux、Image，zImage、uImage 的区别

前面几小节重点是讲 vmlinux 是如何编译出来的，vmlinux 是 ELF 格式的文件，但是在实际中我们不会使用 vmlinux，而是使用 zImage 或 uImage 这样的 Linux 内核镜像文件。那么 vmlinux、zImage、uImage 他们之间有什么区别呢？

①、vmlinux 是编译出来的最原始的内核文件，是未压缩的，比如正点原子提供的 Linux 源码编译出来的 vmlinux 差不多有 16MB，如图 35.5.5.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$ ls vmlinux -l
-rwxrwxr-x 1 zuozhongkai zuozhongkai 16770053 Sep  3 01:44 vmlinux
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$
```

图 35.5.5.1 vmlinux 信息

②、Image 是 Linux 内核镜像文件，但是 Image 仅包含可执行的二进制数据。Image 就是使用 objcopy 取消掉 vmlinux 中的一些其他信息，比如符号表什么的。但是 Image 是没有压缩过的，Image 保存在 arch/arm/boot 目录下，其大小大概在 12MB 左右如图 35.5.5.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$ ls arch/arm/boot/Image -l
-rwxrwxr-x 1 zuozhongkai zuozhongkai 12541952 Sep  3 01:44 arch/arm/boot/Image
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$
```

图 35.5.5.2 Image 镜像信息

相比 vmlinux 的 16MB，Image 缩小到了 12MB。

③、zImage 是经过 gzip 压缩后的 Image，经过压缩以后其大小大概在 6MB 左右，如图 35.5.5.3 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$ ls arch/arm/boot/zImage -l
-rwxrwxr-x 1 zuozhongkai zuozhongkai 6696768 Sep  3 01:44 arch/arm/boot/zImage
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/alientek_linux$
```

图 35.5.5.3 zImage 镜像信息

④、uImage 是老版本 uboot 专用的镜像文件，uImag 是在 zImage 前面加了一个长度为 64 字节的“头”，这个头信息描述了该镜像文件的类型、加载位置、生成时间、大小等信息。但是新的 uboot 已经支持了 zImage 启动!所以已经很少用到 uImage 了，除非你用的很古老的 uboot。

使用“make”、“make all”、“make zImage”这些命令就可以编译出 zImage 镜像，在 arch/arm/Makefile 中有如下代码：

示例代码 35.5.5.1 顶层 Makefile 代码段

```
310 BOOT_TARGETS    = zImage Image xipImage bootpImage uImage
......
315 $(BOOT_TARGETS): vmlinux
```

```
316     $(Q)$(MAKE) $(build)=$(boot) MACHINE=$(MACHINE) $(boot)/$@
```

第 310 行，变量 BOOT_TARGETS 包含 zImage，Image，xipImage 等镜像文件。

第 315 行，BOOT_TARGETS 依赖 vmlinux，因此如果使用“make zImage”编译的 Linux 内核的话，首先肯定要先编译出 vmlinux。

第 316 行，具体的命令，比如要编译 zImage，那么命令展开以后如下所示：

```
@ make -f ./scripts/Makefile.build obj=arch/arm/boot MACHINE=arch/arm/boot/zImage
```

看来又是使用 scripts/Makefile.build 文件来完成 vmliux 到 zImage 的转换。

关于 Linux 顶层 Makefile 就讲解到这里，基本和 uboot 的顶层 Makefile 一样，重点在于 vmlinux 的生成。最后将 vmlinux 压缩成我们最常用的 zImage 或 uImage 等文件。

第三十六章 Linux 内核启动流程

看完 Linux 内核的顶层 Makefile 以后再来看 Linux 内核的大致启动流程，Linux 内核的启动流程要比 uboot 复杂的多，涉及到的内容也更多，因此本章我们就大致的了解一下 Linux 内核的启动流程。

36.1 链接脚本 vmlinux.lds

要分析 Linux 启动流程，同样需要先编译一下 Linux 源码，因为有很多文件是需要编译才会生成的。首先分析 Linux 内核的连接脚本文件 arch/arm/kernel/vmlinux.lds，通过链接脚本可以找到 Linux 内核的第一行程序是从哪里执行的。vmlinux.lds 中有如下代码：

示例代码 36.1.1 vmlinux.lds 链接脚本

```
492 OUTPUT_ARCH(arm)
493 ENTRY(stext)
494 jiffies = jiffies_64;
495 SECTIONS
496 {
497  /*
498      * XXX: The linker does not define how output sections are
499      * assigned to input sections when there are multiple statements
500      * matching the same input section name.  There is no documented
501      * order of matching.
502      *
503      * unwind exit sections must be discarded before the rest of the
504      * unwind sections get included.
505      */
506     /DISCARD/ : {
507     *(.ARM.exidx.exit.text)
508     *(.ARM.extab.exit.text)
509
......
645 }
```

第 493 行的 ENTRY 指明了了 Linux 内核入口，入口为 stext，stext 定义在文件 arch/arm/kernel/head.S 中，因此要分析 Linux 内核的启动流程，就得先从文件 arch/arm/kernel/head.S 的 stext 处开始分析。

36.2 Linux 内核启动流程分析

36.2.1 Linux 内核入口 stext

stext 是 Linux 内核的入口地址，在文件 arch/arm/kernel/head.S 中有如下所示提示内容：

示例代码 36.2.1.1 arch/arm/kernel/head.S 代码段

```
/*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr, r2 = atags or dtb pointer.
 .....
```

```
*/
```

根据示例代码 36.2.1.1 中的注释，Linux 内核启动之前要求如下：

①、关闭 MMU。

②、关闭 D-cache。

③、I-Cache 无所谓。

④、r0=0。

⑤、r1=machine nr(也就是机器 ID)。

⑥、r2=atags 或者设备树(dtb)首地址。

Linux 内核的入口点 stext 其实相当于内核的入口函数，stext 函数内容如下：

示例代码 36.2.1.2 arch/arm/kernel/head.S 代码段

```
80  ENTRY(stext)
......
91      @ ensure svc mode and all interrupts masked
92      safe_svcmode_maskall r9
93
94      mrc p15, 0, r9, c0, c0      @ get processor id
95      bl  __lookup_processor_type     @ r5=procinfo r9=cpuid
96      movs    r10, r5             @ invalid processor (r5=0)?
97   THUMB( it  eq )        @ force fixup-able long branch encoding
98      beq __error_p           @ yes, error 'p'
99
......
107
108 #ifndef CONFIG_XIP_KERNEL
......
113 #else
114     ldr r8, =PLAT_PHYS_OFFSET       @ always constant in this case
115 #endif
116
117     /*
118      * r1 = machine no, r2 = atags or dtb,
119      * r8 = phys_offset, r9 = cpuid, r10 = procinfo
120      */
121     bl  __vet_atags
......
128     bl  __create_page_tables
129
130   /*
131    * The following calls CPU specific code in a position independent
132    * manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
133    * xxx_proc_info structure selected by __lookup_processor_type
134    * above.  On return, the CPU will be ready for the MMU to be
135    * turned on, and r0 will hold the CPU control register value.
```

```
136    */
137    ldr r13, =__mmap_switched       @ address to jump to after
138                          @ mmu has been enabled
139    adr lr, BSYM(1f)           @ return (PIC) address
140    mov r8, r4              @ set TTBR1 to swapper_pg_dir
141    ldr r12, [r10, #PROCINFO_INITFUNC]
142    add r12, r12, r10
143    ret r12
144 1:  b   __enable_mmu
145 ENDPROC(stext)
```

第 92 行，调用函数 safe_svcmode_maskall 确保 CPU 处于 SVC 模式，并且关闭了所有的中断。safe_svcmode_maskall 定义在文件 arch/arm/include/asm/assembler.h 中。

第 94 行，读处理器 ID，ID 值保存在 r9 寄存器中。

第 95 行，调用函数__lookup_processor_type 检查当前系统是否支持此 CPU，如果支持的就获取 procinfo 信息。procinfo 是 proc_info_list 类型的结构体， proc_info_list 在文件 arch/arm/include/asm/procinfo.h 中的定义如下：

示例代码 36.2.1.3 proc_info_list 结构体

```
struct proc_info_list {
    unsigned int          cpu_val;
    unsigned int          cpu_mask;
    unsigned long         __cpu_mm_mmu_flags;  /* used by head.S */
    unsigned long         __cpu_io_mmu_flags;  /* used by head.S */
    unsigned long         __cpu_flush;         /* used by head.S */
    const char            *arch_name;
    const char            *elf_name;
    unsigned int          elf_hwcap;
    const char            *cpu_name;
    struct processor      *proc;
    struct cpu_tlb_fns    *tlb;
    struct cpu_user_fns   *user;
    struct cpu_cache_fns  *cache;
};
```

Linux 内核将每种处理器都抽象为一个 proc_info_list 结构体，每种处理器都对应一个 procinfo。因此可以通过处理器 ID 来找到对应的 procinfo 结构，__lookup_processor_type 函数找到对应处理器的 procinfo 以后会将其保存到 r5 寄存器中。

继续回到示例代码 36.2.1.2 中，第 121 行，调用函数__vet_atags 验证 atags 或设备树(dtb)的合法性。函数__vet_atags 定义在文件 arch/arm/kernel/head-common.S 中。

第 128 行，调用函数__create_page_tables 创建页表。

第 137 行，将函数__mmap_switched 的地址保存到 r13 寄存器中。__mmap_switched 定义在文件 arch/arm/kernel/head-common.S，__mmap_switched 最终会调用 start_kernel 函数。

第 144 行，调用__enable_mmu 函数使能 MMU，__enable_mmu 定义在文件 arch/arm/kernel/head.S 中。__enable_mmu 最终会通过调用__turn_mmu_on 来打开 MMU，__turn_mmu_on 最后会执行 r13 里面保存的__mmap_switched 函数。

36.2.2 __mmap_switched 函数

__mmap_switched 函数定义在文件 arch/arm/kernel/head-common.S 中，函数代码如下：

示例代码 36.2.2.1 __mmap_switched 函数

```
81  __mmap_switched:
82      adr r3, __mmap_switched_data
83  
84      ldmia   r3!, {r4, r5, r6, r7}
85      cmp r4, r5              @ Copy data segment if needed
86  1:  cmpne   r5, r6
87      ldrne   fp, [r4], #4
88      strne   fp, [r5], #4
89      bne 1b
90  
91      mov fp, #0              @ Clear BSS (and zero fp)
92  1:  cmp r6, r7
93      strcc   fp, [r6],#4
94      bcc 1b
95  
96   ARM(   ldmia   r3, {r4, r5, r6, r7, sp})
97   THUMB( ldmia   r3, {r4, r5, r6, r7}    )
98   THUMB( ldr sp, [r3, #16]       )
99      str r9, [r4]            @ Save processor ID
100     str r1, [r5]            @ Save machine type
101     str r2, [r6]            @ Save atags pointer
102     cmp r7, #0
103     strne   r0, [r7]            @ Save control register values
104     b   start_kernel
105 ENDPROC(__mmap_switched)
```

第 104 行最终调用 start_kernel 来启动 Linux 内核，start_kernel 函数定义在文件 init/main.c 中。

36.2.3 start_kernel 函数

start_kernel 通过调用众多的子函数来完成 Linux 启动之前的一些初始化工作，由于 start_kernel 函数里面调用的子函数太多，而这些子函数又很复杂，因此我们简单的来看一下一些重要的子函数。精简并添加注释后的 start_kernel 函数内容如下：

示例代码 36.2.3.1 start_kernel 函数

```
asmlinkage __visible void __init start_kernel(void)
{
    char *command_line;
    char *after_dashes;

    lockdep_init(); /* lockdep 是死锁检测模块，此函数会初始化
```

```
                * 两个 hash 表。此函数要求尽可能早的执行！
                */
set_task_stack_end_magic(&init_task);/* 设置任务栈结束魔术数，
                                       *用于栈溢出检测
                                       */
smp_setup_processor_id();   /* 跟 SMP 有关(多核处理器)，设置处理器 ID。
                             * 有很多资料说 ARM 架构下此函数为空函数，那是因
                             * 为他们用的老版本 Linux，而那时候 ARM 还没有多
                             * 核处理器。
                             */
debug_objects_early_init(); /* 做一些和 debug 有关的初始化 */
boot_init_stack_canary();   /* 栈溢出检测初始化 */
cgroup_init_early(); /* cgroup 初始化，cgroup 用于控制 Linux 系统资源*/
local_irq_disable(); /* 关闭当前 CPU 中断 */
early_boot_irqs_disabled = true;

/*
 * 中断关闭期间做一些重要的操作，然后打开中断
 */
boot_cpu_init();            /* 跟 CPU 有关的初始化 */
page_address_init();        /* 页地址相关的初始化 */
pr_notice("%s", linux_banner);/* 打印 Linux 版本号、编译时间等信息 */
setup_arch(&command_line);  /* 架构相关的初始化，此函数会解析传递进来的
                             * ATAGS 或者设备树(DTB)文件。会根据设备树里面
                             * 的 model 和 compatible 这两个属性值来查找
                             * Linux 是否支持这个单板。此函数也会获取设备树
                             * 中 chosen 节点下的 bootargs 属性值来得到命令
                             * 行参数，也就是 uboot 中的 bootargs 环境变量的
                             * 值，获取到的命令行参数会保存到
                             *command_line 中。
                             */
mm_init_cpumask(&init_mm);  /* 看名字，应该是和内存有关的初始化 */
setup_command_line(command_line);  /* 好像是存储命令行参数 */
setup_nr_cpu_ids();           /* 如果只是 SMP(多核 CPU)的话，此函数用于获取
                               * CPU 核心数量，CPU 数量保存在变量
                               * nr_cpu_ids 中。
                               */
setup_per_cpu_areas(); /* 在 SMP 系统中有用，设置每个 CPU 的 per-cpu 数据 */
smp_prepare_boot_cpu();

build_all_zonelists(NULL, NULL);   /* 建立系统内存页区(zone)链表 */
page_alloc_init();                 /* 处理用于热插拔 CPU 的页 */
```

```
/* 打印命令行信息 */
pr_notice("Kernel command line: %s\n", boot_command_line);
parse_early_param();          /* 解析命令行中的 console 参数 */
after_dashes = parse_args("Booting kernel",
            static_command_line, __start___param,
            __stop___param - __start___param,
            -1, -1, &unknown_bootoption);
if (!IS_ERR_OR_NULL(after_dashes))
    parse_args("Setting init args", after_dashes, NULL, 0, -1, -1,
          set_init_arg);

jump_label_init();

setup_log_buf(0);          /* 设置 log 使用的缓冲区*/
pidhash_init();           /* 构建 PID 哈希表，Linux 中每个进程都有一个 ID,
                           * 这个 ID 叫做 PID。通过构建哈希表可以快速搜索进程
                           * 信息结构体。
                           */
vfs_caches_init_early();       /* 预先初始化 vfs(虚拟文件系统)的目录项和
                               * 索引节点缓存
                               */
sort_main_extable();          /* 定义内核异常列表 */
trap_init();                  /* 完成对系统保留中断向量的初始化 */
mm_init();                    /* 内存管理初始化 */

sched_init();                 /* 初始化调度器，主要是初始化一些结构体 */
preempt_disable();            /* 关闭优先级抢占 */
if (WARN(!irqs_disabled(),    /* 检查中断是否关闭，如果没有的话就关闭中断 */
     "Interrupts were enabled *very* early, fixing it\n"))
    local_irq_disable();
idr_init_cache();             /* IDR 初始化，IDR 是 Linux 内核的整数管理机
                               * 制，也就是将一个整数 ID 与一个指针关联起来。
                               */
rcu_init();    /* 初始化 RCU，RCU 全称为 Read Copy Update(读-拷贝修改) */
trace_init(); /* 跟踪调试相关初始化 */

context_tracking_init();
radix_tree_init();         /* 基数树相关数据结构初始化 */
early_irq_init();          /* 初始中断相关初始化,主要是注册 irq_desc 结构体变
                           * 量，因为 Linux 内核使用 irq_desc 来描述一个中断。
                           */
init_IRQ();                /* 中断初始化 */
tick_init();               /* tick 初始化 */
```

```
    rcu_init_nohz();
    init_timers();              /* 初始化定时器 */
    hrtimers_init();            /* 初始化高精度定时器 */
    softirq_init();             /* 软中断初始化 */
    timekeeping_init();
    time_init();                  /* 初始化系统时间 */
    sched_clock_postinit();
    perf_event_init();
    profile_init();
    call_function_init();
    WARN(!irqs_disabled(), "Interrupts were enabled early\n");
    early_boot_irqs_disabled = false;
    local_irq_enable();        /* 使能中断 */

    kmem_cache_init_late();  /* slab 初始化，slab 是 Linux 内存分配器  */
    console_init();            /* 初始化控制台，之前 printk 打印的信息都存放
                                * 缓冲区中，并没有打印出来。只有调用此函数
                                * 初始化控制台以后才能在控制台上打印信息。
                                */
    if (panic_later)
        panic("Too many boot %s vars at `%s'", panic_later,
              panic_param);

    lockdep_info();/* 如果定义了宏 CONFIG_LOCKDEP，那么此函数打印一些信息。*/

    locking_selftest()         /* 锁自测 */
    ......
    page_ext_init();
    debug_objects_mem_init();
    kmemleak_init();           /* kmemleak 初始化，kmemleak 用于检查内存泄漏 */
    setup_per_cpu_pageset();
    numa_policy_init();
    if (late_time_init)
        late_time_init();
    sched_clock_init();
    calibrate_delay(); /* 测定 BogoMIPS 值，可以通过 BogoMIPS 来判断 CPU 的性能
                        * BogoMIPS 设置越大，说明 CPU 性能越好。
                        */
    pidmap_init();          /* PID 位图初始化 */
    anon_vma_init();        /* 生成 anon_vma slab 缓存 */
    acpi_early_init();
    ......
    thread_info_cache_init();
```

```
    cred_init();            /* 为对象的每个用于赋予资格(凭证)    */
    fork_init();            /* 初始化一些结构体以使用 fork 函数  */
    proc_caches_init();     /* 给各种资源管理结构分配缓存        */
    buffer_init();          /* 初始化缓冲缓存                    */
    key_init();             /* 初始化密钥                        */
    security_init();        /* 安全相关初始化                    */
    dbg_late_init();
    vfs_caches_init(totalram_pages);    /* 为 VFS 创建缓存   */
    signals_init();                     /* 初始化信号        */

    page_writeback_init();              /* 页回写初始化      */
    proc_root_init();                   /* 注册并挂载 proc 文件系统 */
    nsfs_init();
    cpuset_init();          /* 初始化 cpuset，cpuset 是将 CPU 和内存资源以逻辑性
                             * 和层次性集成的一种机制，是 cgroup 使用的子系统之一
                             */
    cgroup_init();                /* 初始化 cgroup */
    taskstats_init_early();       /* 进程状态初始化 */
    delayacct_init();

    check_bugs();                 /* 检查写缓冲一致性 */

    acpi_subsystem_init();
    sfi_init_late();

    if (efi_enabled(EFI_RUNTIME_SERVICES)) {
        efi_late_init();
        efi_free_boot_services();
    }

    ftrace_init();

    rest_init();                  /* rest_init 函数 */
}
```

start_kernel 里面调用了大量的函数，每一个函数都是一个庞大的知识点，如果想要学习 Linux 内核，那么这些函数就需要去详细的研究。本教程注重于嵌入式 Linux 入门，因此不会去讲太多关于 Linux 内核的知识。start_kernel 函数最后调用了 rest_init，接下来简单看一下 rest_init 函数。

36.2.4 rest_init 函数

rest_init 函数定义在文件 init/main.c 中，函数内容如下：

示例代码 36.2.4.1 rest_init 函数

```
383 static noinline void __init_refok rest_init(void)
```

```
384 {
385     int pid;
386 
387     rcu_scheduler_starting();
388     smpboot_thread_init();
389     /*
390      * We need to spawn init first so that it obtains pid 1, however
391      * the init task will end up wanting to create kthreads, which,
392      * if we schedule it before we create kthreadd, will OOPS.
393      */
394     kernel_thread(kernel_init, NULL, CLONE_FS);
395     numa_default_policy();
396     pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);
397     rcu_read_lock();
398     kthreadd_task = find_task_by_pid_ns(pid, &init_pid_ns);
399     rcu_read_unlock();
400     complete(&kthreadd_done);
401 
402     /*
403      * The boot idle thread must execute schedule()
404      * at least once to get things moving:
405      */
406     init_idle_bootup_task(current);
407     schedule_preempt_disabled();
408     /* Call into cpu_idle with preempt disabled */
409     cpu_startup_entry(CPUHP_ONLINE);
410 }
```

第 387 行，调用函数 rcu_scheduler_starting，启动 RCU 锁调度器

第 394 行，调用函数 kernel_thread 创建 kernel_init 线程，也就是大名鼎鼎的 init 内核进程。init 进程的 PID 为 1。init 进程一开始是内核进程(也就是运行在内核态)，后面 init 进程会在根文件系统中查找名为“init”这个程序，这个“init”程序处于用户态，通过运行这个“init”程序，init 进程就会实现从内核态到用户态的转变。

第 396 行，调用函数 kernel_thread 创建 kthreadd 内核进程，此内核进程的 PID 为 2。kthreadd 进程负责所有内核进程的调度和管理。

第 409 行，最后调用函数 cpu_startup_entry 来进入 idle 进程，cpu_startup_entry 会调用 cpu_idle_loop，cpu_idle_loop 是个 while 循环，也就是 idle 进程代码。idle 进程的 PID 为 0，idle 进程叫做空闲进程，如果学过 FreeRTOS 或者 UCOS 的话应该听说过空闲任务。idle 空闲进程就和空闲任务一样，当 CPU 没有事情做的时候就在 idle 空闲进程里面“瞎逛游”，反正就是给 CPU 找点事做。当其他进程要工作的时候就会抢占 idle 进程，从而夺取 CPU 使用权。其实大家应该可以看到 idle 进程并没有使用 kernel_thread 或者 fork 函数来创建，因为它是有主进程演变而来的。

在 Linux 终端中输入“ps -A”就可以打印出当前系统中的所有进程，其中就能看到 init 进程和 kthreadd 进程，如图 36.2.4.1 所示：

```
/ # ps -A
PID   USER       TIME  COMMAND
    1 0          0:01 init
    2 0          0:00 [kthreadd]
    3 0          0:00 [ksoftirqd/0]
    5 0          0:00 [kworker/0:0H]
    7 0          0:00 [rcu_preempt]
```

图 36.2.4.1 Linux 系统当前进程

从图 36.2.4.1 可以看出，init 进程的 PID 为 1，kthreadd 进程的 PID 为 2。之所以图 36.2.4.1 中没有显示 PID 为 0 的 idle 进程，那是因为 idle 进程是内核进程。我们接下来重点看一下 init 进程，kernel_init 就是 init 进程的进程函数。

36.2.5 init 进程

kernel_init 函数就是 init 进程具体做的工作，定义在文件 init/main.c 中，函数内容如下：

示例代码 36.2.5.1 kernel_init 函数

```
928 static int __ref kernel_init(void *unused)
929 {
930     int ret;
931 
932     kernel_init_freeable();         /* init 进程的一些其他初始化工作 */
933     /* need to finish all async __init code before freeing the 
        memory */
934     async_synchronize_full();       /* 等待所有的异步调用执行完成    */
935     free_initmem();                 /* 释放 init 段内存              */
936     mark_rodata_ro();
937     system_state = SYSTEM_RUNNING;  /* 标记系统正在运行              */
938     numa_default_policy();
939 
940     flush_delayed_fput();
941 
942     if (ramdisk_execute_command) {
943         ret = run_init_process(ramdisk_execute_command);
944         if (!ret)
945             return 0;
946         pr_err("Failed to execute %s (error %d)\n",
947               ramdisk_execute_command, ret);
948     }
949 
950     /*
951      * We try each of these until one succeeds.
952      *
953      * The Bourne shell can be used instead of init if we are
954      * trying to recover a really broken machine.
955      */
```

```
956     if (execute_command) {
957         ret = run_init_process(execute_command);
958         if (!ret)
959             return 0;
960         panic("Requested init %s failed (error %d).",
961               execute_command, ret);
962     }
963     if (!try_to_run_init_process("/sbin/init") ||
964         !try_to_run_init_process("/etc/init") ||
965         !try_to_run_init_process("/bin/init") ||
966         !try_to_run_init_process("/bin/sh"))
967         return 0;
968
969     panic("No working init found.  Try passing init= option to 
kernel. "
970           "See Linux Documentation/init.txt for guidance.");
971 }
```

第 932 行，kernel_init_freeable 函数用于完成 init 进程的一些其他初始化工作，稍后再来具体看一下此函数。

第 940 行，ramdisk_execute_command 是一个全局的 char 指针变量，此变量值为“/init”，也就是根目录下的 init 程序。ramdisk_execute_command 也可以通过 uboot 传递，在 bootargs 中使用“rdinit=xxx”即可，xxx 为具体的 init 程序名字。

第 943 行，如果存在“/init”程序的话就通过函数 run_init_process 来运行此程序。

第 956 行，如果 ramdisk_execute_command 为空的话就看 execute_command 是否为空，反正不管如何一定要在根文件系统中找到一个可运行的 init 程序。execute_command 的值是通过 uboot 传递，在 bootargs 中使用“init=xxxx”就可以了，比如“init=/linuxrc”表示根文件系统中的 linuxrc 就是要执行的用户空间 init 程序。

第 963~966 行，如果 ramdisk_execute_command 和 execute_command 都为空，那么就依次查找“/sbin/init”、“/etc/init”、“/bin/init”和“/bin/sh”，这四个相当于备用 init 程序，如果这四个也不存在，那么 Linux 启动失败！

第 969 行，如果以上步骤都没有找到用户空间的 init 程序，那么就提示错误发生！

最后来简单看一下 kernel_init_freeable 函数，前面说了，kernel_init 会调用此函数来做一些 init 进程初始化工作。kernel_init_freeable 定义在文件 init/main.c 中，缩减后的函数内容如下：

示例代码 36.2.5.2 kernel_init_freeable 函数

```
973 static noinline void __init kernel_init_freeable(void)
974 {
975     /*
976      * Wait until kthreadd is all set-up.
977      */
978     wait_for_completion(&kthreadd_done);/* 等待 kthreadd 进程准备就绪 */
......
998
999     smp_init();              /* SMP 初始化           */
```

```
1000    sched_init_smp();       /* 多核(SMP)调度初始化   */
1001
1002    do_basic_setup();      /* 设备初始化都在此函数中完成 */
1003
1004   /* Open the /dev/console on the rootfs, this should never fail */
1005    if (sys_open((const char __user *) "/dev/console", O_RDWR, 0) <
            0)
1006        pr_err("Warning: unable to open an initial console.\n");
1007
1008    (void) sys_dup(0);
1009    (void) sys_dup(0);
1010    /*
1011     * check if there is an early userspace init.  If yes, let it do
1012     * all the work
1013     */
1014
1015    if (!ramdisk_execute_command)
1016        ramdisk_execute_command = "/init";
1017
1018    if (sys_access((const char __user *) ramdisk_execute_command,
            0) != 0) {
1019        ramdisk_execute_command = NULL;
1020        prepare_namespace();
1021    }
1022
1023    /*
1024     * Ok, we have completed the initial bootup, and
1025     * we're essentially up and running. Get rid of the
1026     * initmem segments and start the user-mode stuff..
1027     *
1028     * rootfs is available now, try loading the public keys
1029     * and default modules
1030     */
1031
1032    integrity_load_keys();
1033    load_default_modules();
1034 }
```

第 1002 行，do_basic_setup 函数用于完成 Linux 下设备驱动初始化工作！非常重要。do_basic_setup 会调用 driver_init 函数完成 Linux 下驱动模型子系统的初始化。

第 1005 行，打开设备“/dev/console”，在 Linux 中一切皆为文件！因此“/dev/console”也是一个文件，此文件为控制台设备。每个文件都有一个文件描述符，此处打开的“/dev/console”文件描述符为 0，作为标准输入(0)。

第 1008 和 1009 行，sys_dup 函数将标准输入(0)的文件描述符复制了 2 次，一个作为标准输出(1)，一个作为标准错误(2)。这样标准输入、输出、错误都是/dev/console 了。console 通过 uboot 的 bootargs 环境变量设置，“console=ttymxc0,115200”表示将/dev/ttymxc0 设置为 console，也就是 I.MX6U 的串口 1。当然，也可以设置其他的设备为 console，比如虚拟控制台 tty1，设置 tty1 为 console 就可以在 LCD 屏幕上看到系统的提示信息。

第 1020 行，调用函数 prepare_namespace 来挂载根文件系统。跟文件系统也是由命令行参数指定的，也就是 uboot 的 bootargs 环境变量。比如“root=/dev/mmcblk1p2 rootwait rw”就表示根文件系统在/dev/mmcblk1p2 中，也就是 EMMC 的分区 2 中。

Linux 内核启动流程就分析到这里，Linux 内核最终是需要和根文件系统打交道的，需要挂载根文件系统，并且执行根文件系统中的 init 程序，以此来进去用户态。这里就正式引出了根文件系统，根文件系统也是我们系统移植的最后一片拼图。Linux 移植三巨头：uboot、Linux kernel、rootfs(根文件系统)。关于根文件系统后面章节会详细的讲解，这里我们只需要知道 Linux 内核移植完成以后还需要构建根文件系统即可。

第三十七章 Linux 内核移植

前两章我们简单了解了一下 Linux 内核顶层 Makefile 和 Linux 内核的启动流程，本章我们就来学习一下如何将 NXP 官方提供的 Linux 内核移植到正点原子的 I.MX6U-ALPHA 开发板上。通过本章的学习，我们将掌握如何将半导体厂商提供的 Linux BSP 包移植到我们自己的平台上。

37.1 创建 VSCode 工程

这里我们使用 NXP 官方提供的 Linux 源码，将其移植到正点原子 I.MX6U-ALPHA 开发板上。NXP 官方原版 Linux 源码已经放到了开发板光盘中，路径为：1、例程源码->4、NXP 官方原版 Uboot 和 Linux->linux-imx-rel_imx_4.1.15_2.1.0_ga.tar.bz2。使用 FileZilla 将其发送到 Ubuntu 中并解压，得到名为 linux-imx-rel_imx_4.1.15_2.1.0_ga 的目录，为了和 NXP 官方的名字区分，可以使用"mv"命令对其重命名，我这里将其重命名为"linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek"，命令如下：

```
mv linux-imx-rel_imx_4.1.15_2.1.0_ga linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek
```

完成以后创建 VSCode 工程，步骤和 Windows 下一样，重点是.vscode/settings.json 这个文件。

37.2 NXP 官方开发板 Linux 内核编译

NXP 提供的 Linux 源码肯定是可以在自己的 I.MX6ULL EVK 开发板上运行下去的，所以我们肯定是以 I.MX6ULL EVK 开发板为参考，然后将 Linux 内核移植到 I.MX6U-ALPHA 开发板上的。

37.2.1 修改顶层 Makefile

修改顶层 Makefile，直接在顶层 Makefile 文件里面定义 ARCH 和 CROSS_COMPILE 这两个的变量值为 arm 和 arm-linux-gnueabihf-，结果如图 37.2.1 所示：

```
242  # CROSS_COMPILE specify the prefix used for all executables used
243  # during compilation. Only gcc and related bin-utils executables
244  # are prefixed with $(CROSS_COMPILE).
245  # CROSS_COMPILE can be set on the command line
246  # make CROSS_COMPILE=ia64-linux-
247  # Alternatively CROSS_COMPILE can be set in the environment.
248  # A third alternative is to store a setting in .config so that plain
249  # "make" in the configured kernel build directory always uses that.
250  # Default value for CROSS_COMPILE is not to prefix executables
251  # Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
252  ARCH          ?= arm
253  CROSS_COMPILE    ?= arm-linux-gnueabihf-
```

图 37.2.1 修改顶层 Makefile

图 37.2.1 中第 252 和 253 行分别设置了 ARCH 和 CROSS_COMPILE 这两个变量的值，这样在编译的时候就不用输入很长的命令了。

37.2.2 配置并编译 Linux 内核

和 uboot 一样，在编译 Linux 内核之前要先配置 Linux 内核。每个板子都有其对应的默认配置文件，这些默认配置文件保存在 arch/arm/configs 目录中。imx_v7_defconfig 和 imx_v7_mfg_defconfig 都可作为 I.MX6ULL EVK 开发板所使用的默认配置文件。但是这里建议使用 imx_v7_mfg_defconfig 这个默认配置文件，首先此配置文件默认支持 I.MX6UL 这款芯片，而且重要的一点就是此文件编译出来的 zImage 可以通过 NXP 官方提供的 MfgTool 工具烧写!! imx_v7_mfg_defconfig 中的"mfg"的意思就是 MfgTool。

进入到 Ubuntu 中的 Linux 源码根目录下，执行如下命令配置 Linux 内核：

```
make clean                    //第一次编译 Linux 内核之前先清理一下
```

```
make imx_v7_mfg_defconfig        //配置 Linux 内核
```

配置完成以后如图 37.2.2.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek$ make imx_v7_mfg_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 37.2.2.1 配置 Linux 内核

配置完成以后就可以编译了，使用如下命令编译 Linux 内核：

```
make -j16                        //编译 Linux 内核
```

等待编译完成，结果如图 37.2.2.2 所示：

```
  LD [M]  lib/libcrc32c.ko
  LD [M]  lib/crc-itu-t.ko
  LD [M]  sound/core/snd-rawmidi.ko
  LD [M]  sound/core/snd-hwdep.ko
  LD [M]  sound/usb/snd-usbmidi-lib.ko
  LD [M]  sound/usb/snd-usb-audio.ko
  AS      arch/arm/boot/compressed/piggy.lzo.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek$
```

图 37.2.2.2 Linux 编译完成

Linux 内核编译完成以后会在 arch/arm/boot 目录下生成 zImage 镜像文件，如果使用设备树的话还会在 arch/arm/boot/dts 目录下开发板对应的.dtb(设备树)文件，比如 imx6ull-14x14-evk.dtb 就是 NXP 官方的 I.MX6ULL EVK 开发板对应的设备树文件。至此我们得到两个文件：

①、Linux 内核镜像文件：zImage。

②、NXP 官方 I.MX6ULL EVK 开发板对应的设备树文件：imx6ull-14x14-evk.dtb。

37.2.3 Linux 内核启动测试

在上一小节我们已经得到了 NXP 官方 I.MX6ULL EVK 开发板对应的 zImage 和 imx6ull-14x14-evk.dtb 这两个文件。这两个文件能不能在正点原子的 I.MX6U-ALPHA EMMC 版开发板上启动呢？测试一下不就知道了，在测试之前确保 uboot 中的环境变量 bootargs 内容如下：

```
console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw
```

将上一小节编译出来的 zImage 和 imx6ull-14x14-evk.dtb 复制到 Ubuntu 中的 tftp 目录下，因为我们要在 uboot 中使用 tftp 命令将其下载到开发板中，拷贝命令如下：

```
cp arch/arm/boot/zImage /home/zuozhongkai/linux/tftpboot/ -f
cp arch/arm/boot/dts/imx6ull-14x14-evk.dtb /home/zuozhongkai/linux/tftpboot/ -f
```

拷贝完成以后就可以测试了，启动开发板，进入 uboot 命令行模式，然后输入如下命令将 zImage 和 imx6ull-14x14-evk.dtb 下载到开发板中并启动：

```
tftp 80800000 zImage
tftp 83000000 imx6ull-14x14-evk.dtb
bootz 80800000 – 83000000
```

结果图 37.2.3.1 所示：

```
=> tftp 80800000 zImage
FEC1 Waiting for PHY auto negotiation to complete.... done
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.251
Filename 'zImage'.
Load address: 0x80800000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #
         2.4 MiB/s
done
Bytes transferred = 6680432 (65ef70 hex)
=> tftp 83000000 imx6ull-14x14-evk.dtb
Using FEC1 device
TFTP from server 192.168.1.250; our IP address is 192.168.1.251
Filename 'imx6ull-14x14-evk.dtb'.
Load address: 0x83000000
Loading: ###
         1.2 MiB/s
done
Bytes transferred = 35969 (8c81 hex)
=> bootz 80800000 - 83000000
Kernel image @ 0x80800000 [ 0x000000 - 0x65ef70 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300bc80

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-2017.01)
n 8 12:26:48 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c53c7d
```

图 37.2.3.1 启动 Linux 内核

从图 37.2.3.1 可以看出，此时 Linux 内核已经启动了，如果 EMMC 中的根文件系统存在，我们就可以进入到 Linux 系统里面使用命令进行操作如图 37.2.3.2 所示:

```
VFS: Mounted root (ext3 filesystem) on device 179:10.
devtmpfs: mounted
Freeing unused kernel memory: 440K (80b18000 - 80b86000)
usb 1-1.2: new high-speed USB device number 4 using ci_hdrc
ALSA: Restoring mixer settings...

Please press Enter to activate this console.
/ #
/ #
/ #
/ # random: nonblocking pool is initialized

/ #
/ #
```

图 37.2.3.2 进入 Linux 根文件系统

37.2.4 根文件系统缺失错误

Linux 内核启动以后是需要根文件系统的，根文件系统存在哪里是由 uboot 的 bootargs 环境变量指定，bootargs 会传递给 Linux 内核作为命令行参数。比如上一小节设置 root=/dev/mmcblk1p2，也就是说根文件系统存储在/dev/mmcblk1p2 中，也就是 EMMC 的分区 2 中。这是因为正点原子的 EMMC 版本开发板出厂的时候已经 EMMC 的分区 2 中烧写好了根文件系统，所以设置 root=/dev/mmcblk1p2。如果我们不设置根文件系统路径，或者说根文件系统路径设置错误的话会出现什么问题？这个问题是很常见的，我们在实际的工作中开发一个产品，这个产品的第一版硬件出来以后我们是没有对应的根文件系统可用的，必须要自己做根文件系统。在构建出对应的根文件系统之前 Linux 内核是没有根文件系统可用的，此时 Linux 内核启动以后会出现什么问题呢？带着这个问题，我们将 uboot 中的 bootargs 环境变量改为

“console=ttymxc0,115200”，也就是不填写 root 的内容了，命令如下：

```
setenv bootargs 'console=ttymxc0,115200'
```

修改完成以后重新从网络启动，启动以后会有如图 37.2.4.1 所示错误：

```
usb 1-1.2: new full-speed USB device number 3 using ci_hdrc
mmcblk1boot1: mmc1:0001 4FTE4R partition 2 4.00 MiB
mmcblk1rpmb: mmc1:0001 4FTE4R partition 3 512 KiB
 mmcblk1: p1 p2
VFS: Cannot open root device "(null)" or unknown-block(0,0): error -6
Please append a correct "root=" boot option; here are the available partitions:
0100            65536 ram0  (driver?)
0101            65536 ram1  (driver?)
0102            65536 ram2  (driver?)
0103            65536 ram3  (driver?)
0104            65536 ram4  (driver?)
0105            65536 ram5  (driver?)
0106            65536 ram6  (driver?)
0107            65536 ram7  (driver?)
0108            65536 ram8  (driver?)
0109            65536 ram9  (driver?)
010a            65536 ram10  (driver?)
010b            65536 ram11  (driver?)
010c            65536 ram12  (driver?)
010d            65536 ram13  (driver?)
010e            65536 ram14  (driver?)
010f            65536 ram15  (driver?)
b300         15558144 mmcblk0  driver: mmcblk
  b301          512000 mmcblk0p1 f95b0dec-01
  b302        14943744 mmcblk0p2 f95b0dec-02
b308          3817472 mmcblk1  driver: mmcblk
  b309          512000 mmcblk1p1 b29f4828-01
  b30a         3203072 mmcblk1p2 b29f4828-02
b320              512 mmcblk1rpmb  (driver?)
b318             4096 mmcblk1boot1  (driver?)
b310             4096 mmcblk1boot0  (driver?)
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
```

图 37.2.4.1 根文件系统缺失错误

在图 37.2.4.1 中最后会有下面这一行：

```
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
```

也就是提示内核崩溃，因为 VFS(虚拟文件系统)不能挂载根文件系统，因为根文件系统目录不存在。即使根文件系统目录存在，如果根文件系统目录里面是空的依旧会提示内核崩溃。这个就是根文件系统缺失导致的内核崩溃，但是内核是启动了的，只是根文件系统不存在而已。

37.3 在 Linux 中添加自己的开发板

在 37.2 小节中我们通过编译 NXP 官方 I.MX6ULL EVK 开发板对应的 Linux 内核，发现其可以在正点原子的 EMMC 版本开发板启动，所以我们就参考 I.MX6ULL EVK 开发板的设置，在 Linux 内核中添加正点原子的 I.MX6U-ALPHA 开发板。

37.3.1 添加开发板默认配置文件

将 arch/arm/configs 目录下的 imx_v7_mfg_defconfig 重新复制一份，命名为 imx_alientek_emmc_defconfig，命令如下：

```
cd arch/arm/configs
cp imx_v7_mfg_defconfig imx_alientek_emmc_defconfig
```

以后 imx_alientek_emmc_defconfig 就是正点原子的 EMMC 版开发板默认配置文件了。完成以后如图 37.3.1.1 所示：

```
imx_alientek_emmc_defconfig
imx_v4_v5_defconfig
imx_v6_v7_defconfig
imx_v7_defconfig
imx_v7_mfg_defconfig
```

图 37.3.1.1 新添加的默认配置文件

以后就可以使用如下命令来配置正点原子 EMMC 版开发板对应的 Linux 内核了：

```
make imx_alientek_emmc_defconfig
```

37.3.2 添加开发板对应的设备树文件

添加适合正点原子 EMMC 版开发板的设备树文件，进入目录 arch/arm/boot/dts 中，复制一份 imx6ull-14x14-evk.dts，然后将其重命名为 imx6ull-alientek-emmc.dts，命令如下：

```
cd arch/arm/boot/dts
cp imx6ull-14x14-evk.dts imx6ull-alientek-emmc.dts
```

.dts 是设备树源码文件，编译 Linux 的时候会将其编译为.dtb 文件。imx6ull-alientek-emmc.dts 创建好以后我们还需要修改文件 arch/arm/boot/dts/Makefile，找到“dtb-$(CONFIG_SOC_IMX6ULL)”配置项，在此配置项中加入“imx6ull-alientek-emmc.dtb”，如下所示：

示例代码 37.3.2.1 arch/arm/boot/dts/Makefile 代码段

```
400 dtb-$(CONFIG_SOC_IMX6ULL) += \
401     imx6ull-14x14-ddr3-arm2.dtb \
402     imx6ull-14x14-ddr3-arm2-adc.dtb \
403     imx6ull-14x14-ddr3-arm2-cs42888.dtb \
404     imx6ull-14x14-ddr3-arm2-ecspi.dtb \
405     imx6ull-14x14-ddr3-arm2-emmc.dtb \
406     imx6ull-14x14-ddr3-arm2-epdc.dtb \
407     imx6ull-14x14-ddr3-arm2-flexcan2.dtb \
408     imx6ull-14x14-ddr3-arm2-gpmi-weim.dtb \
409     imx6ull-14x14-ddr3-arm2-lcdif.dtb \
410     imx6ull-14x14-ddr3-arm2-ldo.dtb \
411     imx6ull-14x14-ddr3-arm2-qspi.dtb \
412     imx6ull-14x14-ddr3-arm2-qspi-all.dtb \
413     imx6ull-14x14-ddr3-arm2-tsc.dtb \
414     imx6ull-14x14-ddr3-arm2-uart2.dtb \
415     imx6ull-14x14-ddr3-arm2-usb.dtb \
416     imx6ull-14x14-ddr3-arm2-wm8958.dtb \
417     imx6ull-14x14-evk.dtb \
418     imx6ull-14x14-evk-btwifi.dtb \
419     imx6ull-14x14-evk-emmc.dtb \
420     imx6ull-14x14-evk-gpmi-weim.dtb \
421     imx6ull-14x14-evk-usb-certi.dtb \
422     imx6ull-alientek-emmc.dtb \
```

```
423     imx6ull-9x9-evk.dtb \
424     imx6ull-9x9-evk-btwifi.dtb \
425     imx6ull-9x9-evk-ldo.dtb
```

第 422 行为“imx6ull-alientek-emmc.dtb”，这样编译 Linux 的时候就可以从 imx6ull-alientek-emmc.dts 编译出 imx6ull-alientek-emmc.dtb 文件了。

37.3.3 编译测试

经过 37.3.1 和 37.3.2 两个小节，Linux 内核里面已经添加了正点原子 I.MX6UL-ALIPHA EMMC 版开发板了，接下接编译测试一下，我们可以创建一个编译脚本，imx6ull_alientek_emmc.sh，脚本内容如下：

示例代码 37.3.2.1 imx6ull_alientek_emmc.sh 编译脚本

```
1 #!/bin/sh
2 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- distclean
3 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
imx_alientek_emmc_defconfig
4 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
5 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- all -j16
```

第 2 行，清理工程。

第 3 行，使用默认配置文件 imx_alientek_emmc_defconfig 来配置 Linux 内核。

第 4 行，打开 Linux 的图形配置界面，如果不需要每次都打开图形配置界面可以删除此行。

第 5 行，编译 Linux。

执行 shell 脚本 imx6ull_alientek_emmc.sh 编译 Linux 内核，编译完成以后就会在目录 arch/arm/boot 下生成 zImage 镜像文件。在 arch/arm/boot/dts 目录下生成 imx6ull-alientek-emmc.dtb 文件。将这两个文件拷贝到 tftp 目录下，然后重启开发板，在 uboot 命令模式中使用 tftp 命令下载这两个文件并启动，命令如下：

```
tftp 80800000 zImage
tftp 83000000 imx6ull-alientek-emmc.dtb
bootz 80800000 – 83000000
```

只要出现如图 37.3.3.1 所示内容就表示 Linux 内核启动成功：

```
Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-2017.01) ) #
n 8 18:38:28 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c53c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 ULL 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
Memory policy: Data cache writealloc
```

图 37.3.3.1 Linux 内核启动

Linux 内核启动成功，说明我们已经在 NXP 提供的 Linux 内核源码中添加了正点原子 I.MX6UL-ALPHA 开发板。

37.4 CPU 主频和网络驱动修改

37.4.1 CPU 主频修改

1、设置 I.MX6U-ALPHA 开发板工作在 528MHz

确保 EMMC 中的根文件系统可用！然后重新启动开发板，进入终端(可以输入命令)，如图 37.4.1.1 所示：

```
devtmpfs: mounted
Freeing unused kernel memory: 440K (80b18000 - 80b86000)
usb 1-1.2: device no response, device descriptor read/64, error -32
usb 1-1.2: not running at top speed; connect to a high speed hub
ALSA: Restoring mixer settings...

Please press Enter to activate this console.
/ #
/ #
/ #
/ #
```

图 37.4.1.1 进入命令行

进入图 37.4.1 所示的命令行以后输入如下命令查看 cpu 信息：

```
cat /proc/cpuinfo
```

结果如图 37.4.2 所示：

```
/ # cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 3.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

Hardware        : Freescale i.MX6 Ultralite (Device Tree)
Revision        : 0000
Serial          : 0000000000000000
/ #
```

图 37.4.1.2 cpu 信息

在图 37.4.2.1 中有 BogoMIPS 这一条，此时 BogoMIIS 为 3.00，BogoMIPS 是 Linux 系统中衡量处理器运行速度的一个“尺子”，处理器性能越强，主频越高，BogoMIPS 值就越大。BogoMIPS 只是粗略的计算 CPU 性能，并不十分准确。但是我们可以通过 BogoMIPS 值来大致的判断当前处理器的性能。在图 37.4.1.2 中并没有看到当前 CPU 的工作频率，那我们就转变另一种方法查看当前 CPU 的工作频率。进入到目录/sys/bus/cpu/devices/cpu0/cpufreq 中，此目录下会有很多文件，如图 37.4.1.3 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # ls
affected_cpus                     scaling_cur_freq
cpuinfo_cur_freq                  scaling_driver
cpuinfo_max_freq                  scaling_governor
cpuinfo_min_freq                  scaling_max_freq
cpuinfo_transition_latency        scaling_min_freq
related_cpus                      scaling_setspeed
scaling_available_frequencies     stats
scaling_available_governors
```

图 37.4.1.3 cpufreq 目录

此目录中记录了 CPU 频率等信息，这些文件的含义如下：

cpuinfo_cur_freq：当前 cpu 工作频率，从 CPU 寄存器读取到的工作频率。

cpuinfo_max_freq：处理器所能运行的最高工作频率(单位: KHz）。

cpuinfo_min_freq ：处理器所能运行的最低工作频率(单位: KHz）。

cpuinfo_transition_latency：处理器切换频率所需要的时间(单位:ns)。

scaling_available_frequencies：处理器支持的主频率列表(单位: KHz）。

scaling_available_governors：当前内核中支持的所有 governor(调频)类型。

scaling_cur_freq：保存着 cpufreq 模块缓存的当前 CPU 频率，不会对 CPU 硬件寄存器进行检查。

scaling_driver：该文件保存当前 CPU 所使用的调频驱动。

scaling_governor：governor(调频)策略，Linux 内核一共有 5 中调频策略，

①、Performance，最高性能，直接用最高频率，不考虑耗电。

②、Interactive，一开始直接用最高频率，然后根据 CPU 负载慢慢降低。

③、Powersave，省电模式，通常以最低频率运行，系统性能会受影响，一般不会用这个！

④、Userspace，可以在用户空间手动调节频率。

⑤、Ondemand，定时检查负载，然后根据负载来调节频率。负载低的时候降低 CPU 频率，这样省电，负载高的时候提高 CPU 频率，增加性能。

scaling_max_freq：governor(调频)可以调节的最高频率。

cpuinfo_min_freq：governor(调频)可以调节的最低频率。

stats 目录下给出了 CPU 各种运行频率的统计情况，比如 CPU 在各频率下的运行时间以及变频次数。

使用如下命令查看当前 CPU 频率：

```
cat cpuinfo_cur_freq
```

结果如图 37.4.1.4 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat cpuinfo_cur_freq
198000
/sys/devices/system/cpu/cpu0/cpufreq #
```

图 37.4.1.4 当前 CPU 频率

从图 37.4.1.4 可以看出，当前 CPU 频率为 198MHz，工作频率很低！其他的值如下：

```
cpuinfo_cur_freq = 198000
cpuinfo_max_freq = 528000
cpuinfo_min_freq = 198000
scaling_cur_freq = 198000
scaling_max_freq = 528000
cat scaling_min_freq = 198000
scaling_available_frequencies = 198000 396000 528000
cat scaling_governor = ondemand
```

可以看出，当前 CPU 支持 198MHz、396MHz 和 528Mhz 三种频率切换，其中调频策略为 ondemand，也就是定期检查负载，然后根据负载情况调节 CPU 频率。因为当前我们开发板并没有做什么工作，因此 CPU 频率降低为 198MHz 以省电。如果开发板做一些高负载的工作，比如播放视频、播放视频等操作那么 CPU 频率就会提升上去。查看 stats 目录下的 time_in_state 文件可以看到 CPU 在各频率下的工作时间，命令如下：

```
cat /sys/bus/cpu/devices/cpu0/cpufreq/stats/time_in_state
```

结果如图 37.4.1.5 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq/stats # cat time_in_state
198000 267363
396000 419
528000 2995
/sys/devices/system/cpu/cpu0/cpufreq/stats #
```

图 37.4.1.5 CPU 运行频率统计

从图 37.4.1.5 中可以看出，CPU 在 198MHz、396MHz 和 528MHz 都工作过，其中 198MHz 的工作时间最长！假如我们想让 CPU 一直工作在 528MHz 那该怎么办？很简单，配置 Linux 内

核，将调频策略选择为 performance。或者修改 imx_alientek_emmc_defconfig 文件，此文件中有下面几行：

示例代码 37.4.1.1 调频策略

```
41 CONFIG_CPU_FREQ_DEFAULT_GOV_ONDEMAND=y
42 CONFIG_CPU_FREQ_GOV_POWERSAVE=y
43 CONFIG_CPU_FREQ_GOV_USERSPACE=y
44 CONFIG_CPU_FREQ_GOV_INTERACTIVE=y
```

第 41 行，配置 ondemand 为默认调频策略。

第 42 行，使能 powersave 策略。

第 43 行，使能 userspace 策略。

第 44 行，使能 interactive 策略。

将示例代码 37.4.1.1 中的第 41 行屏蔽掉，然后在 44 行后面添加：

```
CONFIG_CPU_FREQ_GOV_ONDEMAND=y
```

结果下所示：

示例代码 37.4.1.2 修改调频策略

```
41 #CONFIG_CPU_FREQ_DEFAULT_GOV_ONDEMAND=y
42 CONFIG_CPU_FREQ_GOV_POWERSAVE=y
43 CONFIG_CPU_FREQ_GOV_USERSPACE=y
44 CONFIG_CPU_FREQ_GOV_INTERACTIVE=y
45 CONFIG_CPU_FREQ_GOV_ONDEMAND=y
```

修改完成以后重新编译 Linux 内核，编译之前先清理一下工程！因为我们重新修改过默认配置文件了，编译完成以后使用新的 zImage 镜像文件重新启动 Linux。再次查看 /sys/devices/system/cpu/cpu0/cpufreq/ cpuinfo_cur_freq 文件的值，如图 37.4.1.6 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat cpuinfo_cur_freq
528000
/sys/devices/system/cpu/cpu0/cpufreq #
```

图 37.4.1.6 当前 CPU 频率

从图 37.4.1.6 可以看出，当前 CPU 频率为 528MHz 了。查看 scaling_governor 文件看一下当前的调频策略，如图 37.4.1.7 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat scaling_governor
performance
/sys/devices/system/cpu/cpu0/cpufreq #
```

图 37.4.1.7 调频策略

从图 37.4.1.7 可以看出当前的 CPU 调频策略为 preformance，也就是高性能模式，一直以最高主频运行。

我们再来看一下如何通过图形化界面配置 Linux 内核的 CPU 调频策略，输入“make menuconfig”打开 Linux 内核的图形化配置界面，如图 37.4.1.8 所示：

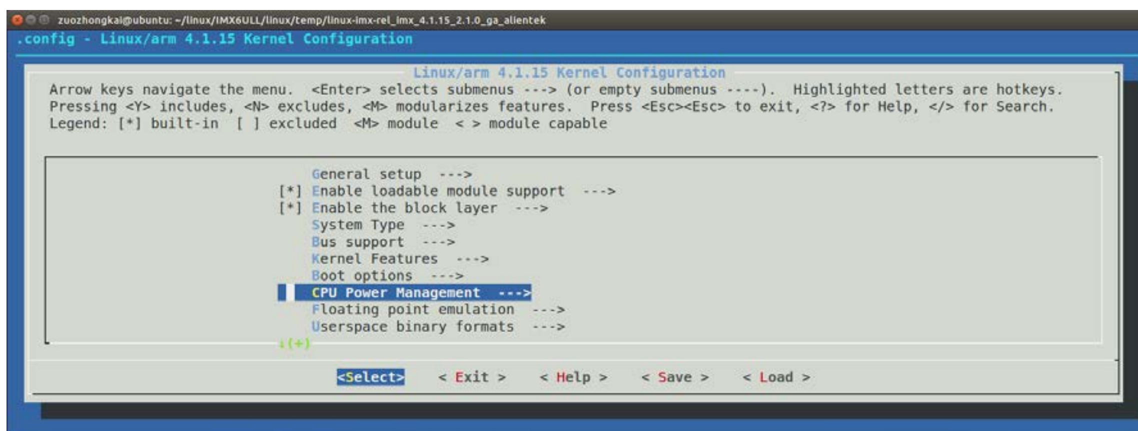


图 37.4.1.8 Linux 内核图形化配置界面

进入如下路径：

```
CPU Power Management
 -> CPU Frequency scaling
   -> CPU Frequency scaling
     -> Default CPUFreq governor
```

打开默认调频策略选择界面，选择“performance”，如图 37.1.4.9 所示：

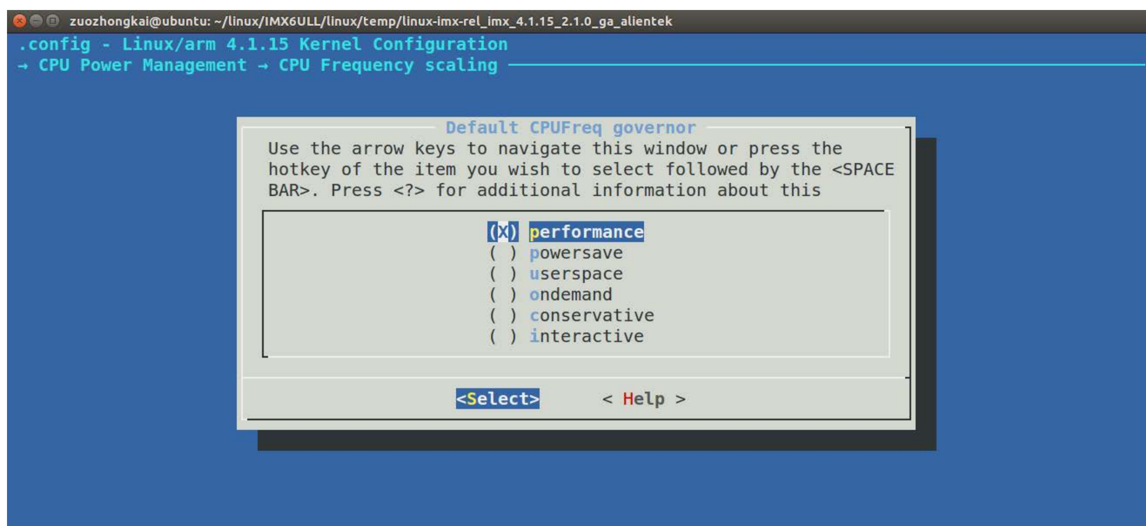


图 37.1.4.9 默认调频策略选择

在图 37.1.4.9 中选择“performance”即可，选择以后退出图形化配置界面，然后编译 Linux 内核，一定不要清理工程！否则的话我们刚刚的设置就会被清理掉。编译完成以后使用新的 zImage 重启 Linux，查看当前 CPU 的工作频率和调频策略。

我们学习的时候为了高性能，大家可以使用 performance 模式。但是在以后的实际产品开发中，从省电的角度考虑，建议大家使用 ondemand 模式，一来可以省电，二来可以减少发热。

2、超频至 700MHz

I.MX6ULL 有多种型号，按照工作频率可以分为 528MHz、700Mhz(实际 696MHz)，800MHz(实际 792MHz)和 900MHz(实际频率未知，应该在 900MHz 左右)。其中 900MHz 的不支持 RGB 屏幕，所以正点原子 Linux 团队没有选择，而 700MHz 的没有工业级(工作温度-40~85°C 以上)，因此也没有选择 700MHz 的型号。最后就剩下了 528MHz 和 800MHz 的，目前正点原子只提供了 528MHz 的版本(型号为 MCIMX6Y2CVM08AB)，至于 800MHz 版本(型号为

MCIMX6Y2CVM08AB)的核心板后续就看客户的需求，如果有需求就会加上，但是 800MHz 版本的 I.MX6ULL 芯片会贵一些，成本会上涨。

虽然正点原子的 I.MX6U-ALPHA 开发板所选择的 I.MX6ULL 标称最高只能工作在 528MHz，但是其是可以超频的 700MHz 的(这里的 700MHz 实际上只有 696MHz，但是 NXP 官方宣传其为 700MHz，所以我们就统一称为 700MHz 吧)。

声明：

对于想体验一下高性能的朋友体验一下超频，虽然笔者一直在用 700MHz 来测试，而且正点原子的 I.MX6U-ALPHA 开发板目前还没有出现过超频不稳定的现象发生，但是！毕竟是超频了的，肯定没有工作在 528MHz 稳定。

如果因为超频带来任何损坏，正点原子不负任何责任！

如果因为超频带来任何损坏，正点原子不负任何责任！

如果因为超频带来任何损坏，正点原子不负任何责任！

在实际的产品中，禁止任何超频！务必严格按照 I.MX6ULL 手册上给出的标准工作频率来运行！！如果想要更高的性能，请购买相应型号的处理器！

看到这里，如果您还是执意要超频，那么就接着往下看，如果要放弃超频，那就跳过本小节，看下一小节。

超频设置其实很简单，修改一下设备树文件 imx6ull.dtsi 即可，打开 imx6ull.dtsi，找到下面代码：

示例代码 37.4.1.3 imx6ull.dtsi 文件代码段

```
54 cpu0: cpu@0 {
55     compatible = "arm,cortex-a7";
56     device_type = "cpu";
57     reg = <0>;
58     clock-latency = <61036>; /* two CLK32 periods */
59     operating-points = <
60         /* kHz  uV */
61         996000  1275000
62         792000  1225000
63         528000  1175000
64         396000  1025000
65         198000  950000
66     >;
67     fsl,soc-operating-points = <
68         /* KHz  uV */
69         996000  1175000
70         792000  1175000
71         528000  1175000
72         396000  1175000
73         198000  1175000
74 >;
```

示例代码 37.4.1.3 就是设置 CPU 频率的，第 61~65 行和第 69~73 行就是 I.MX6ULL 所支持的频率，单位为 KHz，可以看出 I.MX6ULL(视具体型号而定)支持 996MHz、792MHz、528MHz、396MHz 和 198MHz。在上一小节中，我们知道 Linux 内核默认支持 198MHz、396MHz 和 528MHz，

并不能支持到 792MHz，说明 MCIMX6Y2CVM05AB 这颗芯片不能跑到 792MHz。我们在示例代码 37.4.2.1 中加入针对 696MHz 的支持，修改以后代码如下：

示例代码 37.4.1.4 增加 696MHz 的支持

```
54 cpu0: cpu@0 {
55     compatible = "arm,cortex-a7";
56     device_type = "cpu";
57     reg = <0>;
58     clock-latency = <61036>; /* two CLK32 periods */
59     operating-points = <
60         /* kHz  uV */
61         996000  1275000
62         792000  1225000
63         696000  1225000
64         528000  1175000
65         396000  1025000
66         198000  950000
67     >;
68     fsl,soc-operating-points = <
69         /* KHz  uV */
70         996000  1175000
71         792000  1175000
72         696000  1175000
73         528000  1175000
74         396000  1175000
75         198000  1175000
76 >;
```

第 63 行，加入了“696000 1225000”，这个就是 696MHz 的支持。

第 72 行，加入了“696000 1175000”，也是对 696MHz 的支持。

修改好以后保存，并且编译设备树，在 Linux 内核源码根目录下输入如下命令编译设备树：

```
make dtbs
```

命令“make dtbs”只编译设备树文件，也就是将.dts 编译为.dtb，编译完成以后使用新的设备树文件 imx6ull-alientek_emmc.dtb 启动 Linux。重启以后查看文件/sys/devices/system/cpu/cpu0/cpufreq/ scaling_available_frequencies 的内容，如图 37.4.1.10 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat scaling_available_frequencies
198000 396000 528000 696000
/sys/devices/system/cpu/cpu0/cpufreq #
```

图 37.4.1.10 文件 scaling_available_frequencies 内容

从图 37.4.1.11 可以看出，此时支持了 696MHz。如果设置调频策略为 performance，那么处理器就会一直工作在 696MHz。可以对比一下工作在 528MHz 和 696MHz 下的 BogoMIPS 的值，528MHz 主频下的 BogoMIPS 值如图 37.4.1.12 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 8.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

Hardware        : Freescale i.MX6 Ultralite (Device Tree)
Revision        : 0000
Serial          : 0000000000000000
/sys/devices/system/cpu/cpu0/cpufreq #
```

图 37.4.1.12 528MHz 主频下的 BogoMIPS

696MHz 主频下的 BogoMIPS 值如图 37.4.1.13 所示：

```
/sys/devices/system/cpu/cpu0/cpufreq # cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 5 (v7l)
BogoMIPS        : 10.54
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xc07
CPU revision    : 5

Hardware        : Freescale i.MX6 Ultralite (Device Tree)
Revision        : 0000
Serial          : 0000000000000000
```

图 37.4.1.13 696MHz 主频下的 BogoMIPS

从图 37.4.1.12 和图 37.2.1.13 中可以看到，528MHz 和 696MHz 下的 BogoMIPS 值分别为 8.00 和 10.54，相当于性能提升了(10.54/8)-1=31.75%。

37.4.2 使能 8 线 EMMC 驱动

正点原子 EMMC 版本核心板上的 EMMC 采用的 8 位数据线，原理图如图 37.4.2.1 所示：

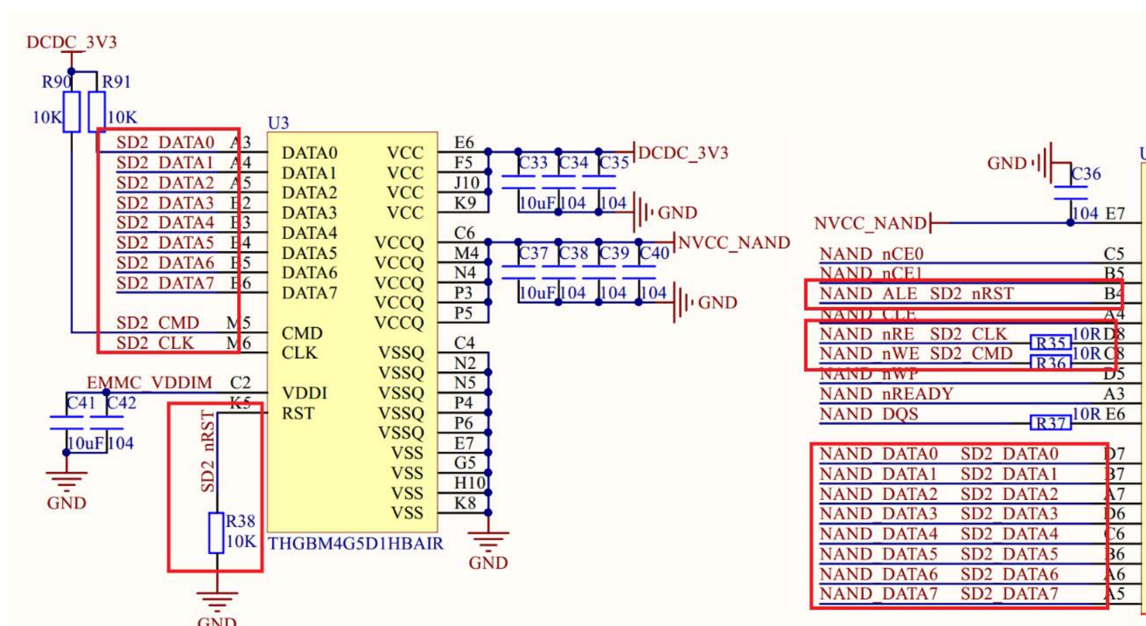


图 37.4.2.1 EMMC 原理图

Linux 内核驱动里面 EMMC 默认是 4 线模式的，4 线模式肯定没有 8 线模式的速度快，所以本节我们将 EMMC 的驱动修改为 8 线模式。修改方法很简单，直接修改设备树即可，打开文件 imx6ull-alientek-emmc.dts，找到如下所示内容：

示例代码 37.4.2.1 imx6ull-alientek-emmc.dts 代码段

```
734 &usdhc2 {
735     pinctrl-names = "default";
736     pinctrl-0 = <&pinctrl_usdhc2>;
737     non-removable;
738     status = "okay";
739 };
```

关于设备树的原理以及内容我们后面会有专门的章节讲解，示例代码 37.4.2.1 中的代码含义我们现在不去纠结，只需要将其改为如下代码即可：

示例代码 37.4.2.1 imx6ull-alientek-emmc.dts 代码段

```
734 &usdhc2 {
735     pinctrl-names = "default", "state_100mhz", "state_200mhz";
736     pinctrl-0 = <&pinctrl_usdhc2_8bit>;
737     pinctrl-1 = <&pinctrl_usdhc2_8bit_100mhz>;
738     pinctrl-2 = <&pinctrl_usdhc2_8bit_200mhz>;
739     bus-width = <8>;
740     non-removable;
741     status = "okay";
742 };
```

修改完成以后保存一下 imx6ull-alientek-emmc.dts，然后使用命令“make dtbs”重新编译一下设备树，编译完成以后使用新的设备树重启 Linux 系统即可。

37.4.3 修改网络驱动

因为在后面学习 Linux 驱动开发的时候要用到网络调试驱动，所以必须要把网络驱动调试好。在讲解 uboot 移植的时候就已经说过了，正点原子开发板的网络和 NXP 官方的网络硬件上不同，网络 PHY 芯片由 KSZ8081 换为了 LAN8720A，两个网络 PHY 芯片的复位 IO 也不同。所以 Linux 内核自带的网络驱动是驱动不起来 I.MX6U-ALPHA 开发板上的网络的，需要做修改。

1、修改 LAN8720 的复位引脚驱动

ENET1 复位引脚 ENET1_RST 连接在 I.M6ULL 的 SNVS_TAMPER7 这个引脚上。ENET2 的复位引脚 ENET2_RST 连接在 I.MX6ULL 的 SNVS_TAMPER8 上。打开设备树文件 imx6ull-alientek-emmc.dts，找到如下代码：

示例代码 37.4.3.1 imx6ull-alientek-emmc.dts 代码段

```
584 pinctrl_spi4: spi4grp {
585             fsl,pins = <
586                 MX6ULL_PAD_BOOT_MODE0__GPIO5_IO10       0x70a1
587                 MX6ULL_PAD_BOOT_MODE1__GPIO5_IO11       0x70a1
588                 MX6ULL_PAD_SNVS_TAMPER7__GPIO5_IO07     0x70a1
589                 MX6ULL_PAD_SNVS_TAMPER8__GPIO5_IO08     0x80000000
590             >;
591         };
```

示例代码 37.4.3.1 中第 588 和 589 行就是初始化 SNVS_TAMPER7 和 SNVS_TAMPER8 这两个

引脚的，不过看样子好像是作为了 SPI4 的 IO，这不是我们想要的，所以将 588 和 589 这两行删除掉！删除掉以后继续在 imx6ull-alientek-emmc.dts 中找到如下所示代码：

示例代码 37.4.3.2 imx6ull-alientek-emmc.dts 代码段

```
125 spi4 {
126     compatible = "spi-gpio";
127     pinctrl-names = "default";
128     pinctrl-0 = <&pinctrl_spi4>;
129     pinctrl-assert-gpios = <&gpio5 8 GPIO_ACTIVE_LOW>;
......
133     cs-gpios = <&gpio5 7 0>;
```

第 129 行，设置 GPIO5_IO08 为 SPI4 的一个功能引脚(我也不清楚具体作为什么功能用)，而 GPIO5_IO08 就是 SNVS_TAMPER8 的 GPIO 功能引脚。

第 133 行，设置 GPIO5_IO07 作为 SPI4 的片选引脚，而 GPIO5_IO07 就是 SNVS_TAMPER7 的 GPIO 功能引脚。

现在我们需要 GPIO5_IO07 和 GPIO5_IO08 分别作为 ENET1 和 ENET2 的复位引脚，而不是 SPI4 的什么功能引脚，因此将示例代码 37.4.3.2 中的第 129 行和第 133 行处的代码删除掉！！否则会干扰到网络复位引脚！

继续在 imx6ull-alientek-emmc.dts 中找到如下所示代码：

示例代码 37.4.3.3 imx6ull-alientek-emmc.dts 代码段

```
309 pinctrl_enet1: enet1grp {
310     fsl,pins = <
311         MX6UL_PAD_ENET1_RX_EN__ENET1_RX_EN          0x1b0b0
312         MX6UL_PAD_ENET1_RX_ER__ENET1_RX_ER          0x1b0b0
313         MX6UL_PAD_ENET1_RX_DATA0__ENET1_RDATA00     0x1b0b0
314         MX6UL_PAD_ENET1_RX_DATA1__ENET1_RDATA01     0x1b0b0
315         MX6UL_PAD_ENET1_TX_EN__ENET1_TX_EN          0x1b0b0
316         MX6UL_PAD_ENET1_TX_DATA0__ENET1_TDATA00     0x1b0b0
317         MX6UL_PAD_ENET1_TX_DATA1__ENET1_TDATA01     0x1b0b0
318         MX6UL_PAD_ENET1_TX_CLK__ENET1_REF_CLK1      0x4001b031
319     >;
320 };
321 
322 pinctrl_enet2: enet2grp {
323     fsl,pins = <
324         MX6UL_PAD_GPIO1_IO07__ENET2_MDC             0x1b0b0
325         MX6UL_PAD_GPIO1_IO06__ENET2_MDIO            0x1b0b0
326         MX6UL_PAD_ENET2_RX_EN__ENET2_RX_EN          0x1b0b0
327         MX6UL_PAD_ENET2_RX_ER__ENET2_RX_ER          0x1b0b0
328         MX6UL_PAD_ENET2_RX_DATA0__ENET2_RDATA00     0x1b0b0
329         MX6UL_PAD_ENET2_RX_DATA1__ENET2_RDATA01     0x1b0b0
330         MX6UL_PAD_ENET2_TX_EN__ENET2_TX_EN          0x1b0b0
331         MX6UL_PAD_ENET2_TX_DATA0__ENET2_TDATA00     0x1b0b0
332         MX6UL_PAD_ENET2_TX_DATA1__ENET2_TDATA01     0x1b0b0
```

```
333         MX6UL_PAD_ENET2_TX_CLK__ENET2_REF_CLK2     0x4001b031
334     >;
335 };
```

第 309~320 行，pinctrl_enet1 是 ENET1 的 IO 初始化配置。

第 322~335 行，pinctrl_enet2 是 ENET2 的 IO 初始化配置。

根据示例代码 37.4.3.3，我们需要将 ENET1 的复位 IO 初始化配置添加到 pinctrl_enet1 中，将 ENET2 的复位 IO 初始化配置添加到 pinctrl_enet2 中，添加完成以后的代码如下所示：

示例代码 37.4.3.4 imx6ull-alientek-emmc.dts 代码段

```
309 pinctrl_enet1: enet1grp {
310     fsl,pins = <
311         MX6UL_PAD_ENET1_RX_EN__ENET1_RX_EN         0x1b0b0
312         MX6UL_PAD_ENET1_RX_ER__ENET1_RX_ER         0x1b0b0
...
318         MX6UL_PAD_ENET1_TX_CLK__ENET1_REF_CLK1     0x4001b031
319         MX6UL_PAD_SNVS_TAMPER7__GPIO5_IO07    0x10B0/* ENET1 RESET */
320     >;
321 };
322
323 pinctrl_enet2: enet2grp {
324     fsl,pins = <
325         MX6UL_PAD_GPIO1_IO07__ENET2_MDC            0x1b0b0
326         MX6UL_PAD_GPIO1_IO06__ENET2_MDIO           0x1b0b0
...
334         MX6UL_PAD_ENET2_TX_CLK__ENET2_REF_CLK2     0x4001b031
335         MX6UL_PAD_SNVS_TAMPER8__GPIO5_IO08  0x10B0 /* ENET2 RESET */
336     >;
337 };
```

第 319 行，ENET1 复位引脚 SNVS_TAMPER7 的配置代码。

第 335 行，ENET2 复位引脚 SNVS_TAMPER8 的配置代码。

修改完成以后记得保存一下 imx6ull-alientek-emmc.dts，网络的复位引脚驱动就修改好了。

2、修改 LAN8720A 的 PHY 地址

在 uboot 移植章节中，我们说过 ENET1 的 LAN8720A 地址为 0x0，ENET2 的 LAN8720A 地址为 0x1。在 imx6ull-alientek-emmc.dts 中找到如下代码：

示例代码 37.4.3.5 imx6ull-alientek-emmc.dts 代码段

```
171 &fec1 {
172     pinctrl-names = "default";
173     pinctrl-0 = <&pinctrl_enet1>;
174     phy-mode = "rmii";
175     phy-handle = <&ethphy0>;
176     status = "okay";
177 };
178
179 &fec2 {
```

```
180     pinctrl-names = "default";
181     pinctrl-0 = <&pinctrl_enet2>;
182     phy-mode = "rmii";
183     phy-handle = <&ethphy1>;
184     status = "okay";
185 
186     mdio {
187         #address-cells = <1>;
188         #size-cells = <0>;
189 
190         ethphy0: ethernet-phy@0 {
191             compatible = "ethernet-phy-ieee802.3-c22";
192             reg = <2>;
193         };
194 
195         ethphy1: ethernet-phy@1 {
196             compatible = "ethernet-phy-ieee802.3-c22";
197             reg = <1>;
198         };
199     };
200 };
```

第 171~177 行，ENET1 对应的设备树节点。

第 179~200 行，ENET2 对应的设备树节点。但是第 186~198 行的 mdio 节点描述了 ENET1 和 ENET2 的 PHY 地址信息。将示例代码 37.4.3.5 改为如下内容：

示例代码 37.4.3.6 imx6ull-alientek-emmc.dts 代码段

```
171 &fec1 {
172     pinctrl-names = "default";
173     pinctrl-0 = <&pinctrl_enet1>;
174     phy-mode = "rmii";
175     phy-handle = <&ethphy0>;
176     phy-reset-gpios = <&gpio5 7 GPIO_ACTIVE_LOW>;
177     phy-reset-duration = <26>;
178     status = "okay";
179 };
180 
181 &fec2 {
182     pinctrl-names = "default";
183     pinctrl-0 = <&pinctrl_enet2>;
184     phy-mode = "rmii";
185     phy-handle = <&ethphy1>;
186     phy-reset-gpios = <&gpio5 8 GPIO_ACTIVE_LOW>;
187     phy-reset-duration = <26>;
188     status = "okay";
```

```
189
190     mdio {
191         #address-cells = <1>;
192         #size-cells = <0>;
193
194         ethphy0: ethernet-phy@0 {
195             compatible = "ethernet-phy-ieee802.3-c22";
196             smsc,disable-energy-detect;
197             reg = <0>;
198          };
199
200         ethphy1: ethernet-phy@1 {
201             compatible = "ethernet-phy-ieee802.3-c22";
202             smsc,disable-energy-detect;
203             reg = <1>;
204         };
205     };
206 };
```

第 176 和 177 行，添加了 ENET1 网络复位引脚所使用的 IO 为 GPIO5_IO07，低电平有效。复位低电平信号持续时间为 26ms。

第 186 和 187 行，ENET2 网络复位引脚所使用的 IO 为 GPIO5_IO08，同样低电平有效，持续时间同样为 26ms。

第 196 和 202 行，“smsc,disable-energy-detect”表明 PHY 芯片是 SMSC 公司的，这样 Linux 内核就会找到 SMSC 公司的 PHY 芯片驱动来驱动 LAN8720A。

第 194 行，注意“ethernet-phy@”后面的数字是 PHY 的地址，ENET1 的 PHY 地址为 0，所以“@”后面是 0(默认为 2)。

第 197 行，reg 的值也表示 PHY 地址，ENET1 的 PHY 地址为 0，所以 reg=0。

第 200 行，ENET2 的 PHY 地址为 1，因此“@”后面为 1。

第 203 行，因为 ENET2 的 PHY 地址为 1，所以 reg=1。

至此，LAN8720A 的 PHY 地址就改好了，保存一下 imx6ull-alientek-emmc.dts 文件。然后使用“make dtbs”命令重新编译一下设备树。

3、修改 fec_main.c 文件

要在 I.MX6ULL 上使用 LAN8720A，需要修改一下 Linux 内核源码，打开 drivers/net/ethernet/freescale/fec_main.c，找到函数 fec_probe，在 fec_probe 中加入如下代码：

示例代码 37.4.3.7 imx6ull-alientek-emmc.dts 代码段

```
3438 static int
3439 fec_probe(struct platform_device *pdev)
3440 {
3441     struct fec_enet_private *fep;
3442     struct fec_platform_data *pdata;
3443     struct net_device *ndev;
3444     int i, irq, ret = 0;
3445     struct resource *r;
```

```
3446    const struct of_device_id *of_id;
3447    static int dev_id;
3448    struct device_node *np = pdev->dev.of_node, *phy_node;
3449    int num_tx_qs;
3450    int num_rx_qs;
3451
3452    /* 设置 MX6UL_PAD_ENET1_TX_CLK 和 MX6UL_PAD_ENET2_TX_CLK
3453     * 这两个 IO 的复用寄存器的 SION 位为 1。
3454     */
3455     void __iomem *IMX6U_ENET1_TX_CLK;
3456     void __iomem *IMX6U_ENET2_TX_CLK;
3457
3458     IMX6U_ENET1_TX_CLK = ioremap(0X020E00DC, 4);
3459     writel(0X14, IMX6U_ENET1_TX_CLK);
3460
3461     IMX6U_ENET2_TX_CLK = ioremap(0X020E00FC, 4);
3462     writel(0X14, IMX6U_ENET2_TX_CLK);
3463
......
3656    return ret;
3657 }
```

第 3455~3462 就是新加入的代码，如果要在 I.MX6ULL 上使用 LAN8720A 就需要设置 ENET1 和 ENET2 的 TX_CLK 引脚复位寄存器的 SION 位为 1。

4、配置 Linux 内核，使能 LAN8720 驱动

输入命令“make menuconfig”，打开图形化配置界面，选择使能 LAN8720A 的驱动，路径如下：

```
-> Device Drivers
    -> Network device support
        -> PHY Device support and infrastructure
            -> Drivers for SMSC PHYs
```

如图 37.4.3.1 所示：

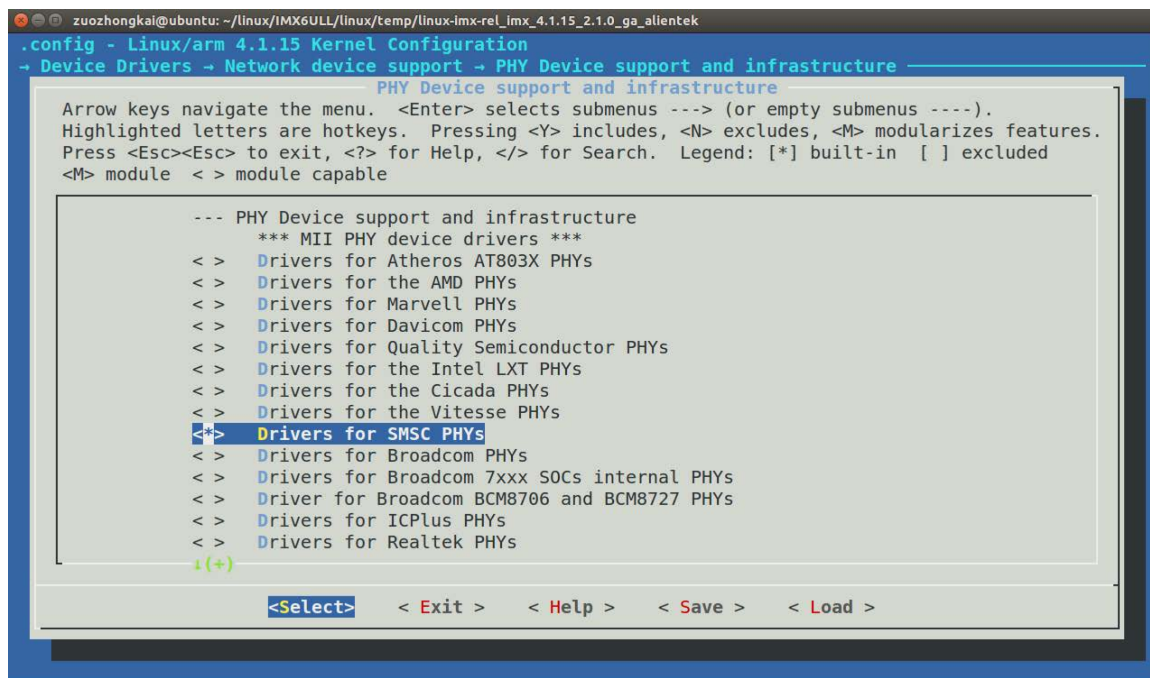


图 37.4.3.1 使能 LAN8720A 驱动

图 37.4.3.1 中选择将“Drivers for SMSC PHYs”编译到 Linux 内核中，因此“<>”里面变为了“*”。LAN8720A 是 SMSC 公司出品的，因此勾选这个以后就会编译 LAN8720 驱动，配置好以后退出配置界面，然后重新编译一下 Linux 内核。

5、修改 smsc.c 文件

在修改 smsc.c 文件之前先说点题外话，那就是我是怎么确定要修改 smsc.c 这个文件的。在写本书之前我并没有修改过 smsc.c 这个文件，都是使能 LAN8720A 驱动以后就直接使用。但是我在测试 NFS 挂载文件系统的时候发现文件系统挂载成功率很低！老是提示 NFS 服务器找不到，三四次就有一次挂载失败！很折磨人。NFS 挂载就是通过网络来挂载文件系统，这样做的好处就是方便我们后续调试 Linux 驱动。既然老是挂载失败那么可以肯定的是网络驱动有问题，网络驱动分两部分：内部 MAC+外部 PHY，内部 MAC 驱动是由 NXP 提供的，一般不会出问题，否则的话用户早就给 NXP 反馈了。而且我用 NXP 官方的开发板测试网络是一直正常的，但是 NXP 官方的开发板所使用的 PHY 芯片为 KSZ8081。所以只有可能是外部 PHY，也就是 LAN8720A 的驱动可能出问题了。鉴于 LAN8720A 有“前车之鉴”，那就是在 uboot 中需要对 LAN8720A 进行一次软复位，要设置 LAN8720A 的 BMCR(寄存器地址为 0)寄存器 bit15 为 1。所以我猜测，在 Linux 中也需要对 LAN8720A 进行一次软复位。

首先需要找到 LAN8720A 的驱动文件，LAN8720A 的驱动文件是 drivers/net/phy/smsc.c，在此文件中有个叫做 smsc_phy_reset 的函数，看名字都知道这是 SMSC PHY 的复位函数，因此，LAN8720A 肯定也会使用到这个复位函数，此函数内容如下：

示例代码 37.4.3.8 smsc_phy_reset 函数

```
60 static int smsc_phy_reset(struct phy_device *phydev)
61 {
62     int rc = phy_read(phydev, MII_LAN83C185_SPECIAL_MODES);
63     if (rc < 0)
64         return rc;
65
```

```
66      /* If the SMSC PHY is in power down mode, then set it
67       * in all capable mode before using it.
68       */
69      if ((rc & MII_LAN83C185_MODE_MASK) ==
MII_LAN83C185_MODE_POWERDOWN) {
70          int timeout = 50000;
71
72          /* set "all capable" mode and reset the phy */
73          rc |= MII_LAN83C185_MODE_ALL;
74          phy_write(phydev, MII_LAN83C185_SPECIAL_MODES, rc);
75          phy_write(phydev, MII_BMCR, BMCR_RESET);
76
77          /* wait end of reset (max 500 ms) */
78          do {
79              udelay(10);
80              if (timeout-- == 0)
81                  return -1;
82              rc = phy_read(phydev, MII_BMCR);
83          } while (rc & BMCR_RESET);
84      }
85      return 0;
86 }
```

第 69 行，只有 PHY 处于 power down 模式的时候第 70~83 行的代码段才会执行。

第 75 行，向 LAN872A0 的 BMCR 寄存器写入 BMCR_RESET，也就是对 LAN8720A 进行软件初始化，所以 smsc_phy_reset 函数会对 LAN8720A 进行软件初始化。

看到没，只有 LAN8720A 处于 power down 模式的时候才会对 LAN8720A 进行软复位，但是我们在 uboot 中已经“唤醒”了 LAN8720A，LAN8720A 也已经工作了，因此它不可能处于 power down 模式，进而就没法对 LAN8720A 进行软复位。因此，我们要对 smsc_phy_reset 函数函数进行修改，将复位相关的代码从条件语句中提出来，不管 LAN8720A 有没有工作在 power down 模式下，只要调用 smsc_phy_reset 函数就对 LAN8720A 进行软复位，修改后的 smsc_phy_reset 函数内容如下：

示例代码 37.4.3.9 修改后的 smsc_phy_reset 函数

```
60 static int smsc_phy_reset(struct phy_device *phydev)
61 {
62      int rc = phy_read(phydev, MII_LAN83C185_SPECIAL_MODES);
63      if (rc < 0)
64          return rc;
65
66      /* If the SMSC PHY is in power down mode, then set it
67       * in all capable mode before using it.
68       */
69      if ((rc & MII_LAN83C185_MODE_MASK) ==
MII_LAN83C185_MODE_POWERDOWN) {
```

```
70
71          /* set "all capable" mode and reset the phy */
72          rc |= MII_LAN83C185_MODE_ALL;
73          phy_write(phydev, MII_LAN83C185_SPECIAL_MODES, rc);
74      }
75
76      phy_write(phydev, MII_BMCR, BMCR_RESET);
77      /* wait end of reset (max 500 ms) */
78      int timeout = 50000;
79      do {
80          udelay(10);
81          if (timeout-- == 0)
82              return -1;
83          rc = phy_read(phydev, MII_BMCR);
84      } while (rc & BMCR_RESET);
85
86      return 0;
87 }
```

重点是 76~84 行，我们将软复位代码移出来，这样每次调用 smsc_phy_reset 函数 LAN8720A 都会被软复位。修改以后基本上每次通过 NFS 挂载根文件系统都会成功。

6、网络驱动测试

修改好设备树和 Linux 内核以后重新编译一下，得到新的 zImage 镜像文件和 imx6ull-alientek-emmc.dtb 设备树文件，使用网线将 I.MX6U-ALPHA 开发板的两个网口与路由器或者电脑连接起来，最后使用新的文件启动 Linux 内核。启动以后使用“ifconfig”命令查看一下当前活动的网卡有哪些，结果如图 37.4.3.2 所示：

```
Please press Enter to activate this console.
/ # ifconfig
/ #
```

图 37.4.3.2 ifconfig 命令结果

从图 37.4.3.2 可以看出，当前没有活动的网卡。输入命令“ifconfig -a”来查看一下开发板中存在的所有网卡，结果如图 37.4.3.3 所示：

```
/ # ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:27

can1      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:28

eth0      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          LOOPBACK  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

图 37.4.3.3 开发板所有网卡

图 37.4.3.3 中 can0 和 can1 为 CAN 接口的网卡，eth0 和 eth1 才是网络接口的网卡，其中 eth0 对应于 ENET1，eth1 对应于 ENET2。使用如下命令依次打开 eth0 和 eth1 这两个网卡：

```
ifconfig eth0 up
ifconfig eth1 up
```

网卡的打开过程如图 37.4.3.4 所示：

```
/ # ifconfig eth0 up
fec 20b4000.ethernet eth0: Freescale FEC PHY driver [SMSC LAN8710/LAN8720] (mii_bus:phy_addr=20b4000.ethernet
:01, irq=-1)
/ # fec 20b4000.ethernet eth0: Link is Up - 100Mbps/Full - flow control rx/tx

/ # ifconfig eth1 up
fec 2188000.ethernet eth1: Freescale FEC PHY driver [SMSC LAN8710/LAN8720] (mii_bus:phy_addr=20b4000.ethernet
:00, irq=-1)
IPv6: ADDRCONF(NETDEV_UP): eth1: link is not ready
/ # fec 2188000.ethernet eth1: Link is Up - 100Mbps/Full - flow control rx/tx
IPv6: ADDRCONF(NETDEV_CHANGE): eth1: link becomes ready
IPv6: eth1: IPv6 duplicate address fe80::204:9fff:fe04:d235 detected!

/ #
```

图 37.4.3.4 两个网卡打开过程

从图 37.4.3.4 中可以看到“SMSC LAN8710/LAN8720”字样，说明当前的网络驱动使用的就是我们前面使能的 SMSC 驱动。

再次输入“ifconfig”命令来查看一下当前活动的网卡，结果如图 37.4.3.5 所示：

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          inet6 addr: fe80::204:9fff:fe04:d235/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1643 errors:0 dropped:19 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:110575 (107.9 KiB)  TX bytes:992 (992.0 B)

eth1      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          inet6 addr: fe80::204:9fff:fe04:d235/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1394 errors:0 dropped:18 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:94453 (92.2 KiB)  TX bytes:258 (258.0 B)
```

图 37.4.3.5 当前活动的网卡。

可以看出，此时 eth0 和 eth1 两个网卡都已经打开，并且工作正常，但是这两个网卡都还没有 IP 地址，所以不能进行 ping 等操作。使用如下命令给两个网卡配置 IP 地址：

```
ifconfig eth0 192.168.1.251
ifconfig eth1 192.168.1.252
```

上述命令配置 eth0 和 eth1 这两个网卡的 IP 地址分别为 192.168.1.251 和 192.168.1.252，注意 IP 地址选择的合理性，一定要和自己的电脑处于同一个网段内，并且没有被其他的设备占用！设置好以后，使用“ping”命令来 ping 一下自己的主机，如果能 ping 通那说明网络驱动修改成功！比如我的 Ubuntu 主机 IP 地址为 192.168.1.250，使用如下命令 ping 一下：

```
ping 192.168.1.250
```

结果如图 37.4.3.6 所示：

```
/ # ping 192.168.1.250
PING 192.168.1.250 (192.168.1.250): 56 data bytes
64 bytes from 192.168.1.250: seq=0 ttl=64 time=0.906 ms
64 bytes from 192.168.1.250: seq=1 ttl=64 time=1.016 ms
64 bytes from 192.168.1.250: seq=2 ttl=64 time=0.893 ms
^C
--- 192.168.1.250 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.893/0.938/1.016 ms
/ #
```

图 37.4.3.6 ping 结果

可以看出，ping 成功，说明网络驱动修改成功！我们在后面的构建根文件系统和 Linux 驱动开发中就可以使用网络调试代码啦，好嗨森！

37.4.4 保存修改后的图形化配置文件

在修改网络驱动的时候我们通过图形界面使能了 LAN8720A 的驱动，使能以后会在.config 中存在如下代码：

```
CONFIG_SMSC_PHY=y
```

打开 drivers/net/phy/Makefile，有如下代码：

示例代码 37.4.4.1 drivers/net/phy/Makefile 代码段

```
11 obj-$(CONFIG_SMSC_PHY)      += smsc.o
```

当 CONFIG_SMSC_PHY=y 的时候就会编译 smsc.c 这个文件，smsc.c 就是 LAN8720A 的驱动文件。但是当我们执行“make clean”清理工程以后.config 文件就会被删除掉，因此我们所有的配置内容都会丢失，结果就是前功尽弃，一“删”回到解放前！所以我们在配置完图形界面以后经过测试没有问题，就必须保存一下配置文件。保存配置的方法有两个。

1、直接另存为.config 文件

既然图形化界面配置后的配置项保存在.config 中，那么就简单粗暴，直接将.config 文件另存为 imx_alientek_emmc_defconfig，然后其复制到 arch/arm/configs 目录下，替换以前的 imx_alientek_emmc_defconfig。这样以后执行“make imx_alientek_emmc_defconfig”重新配置 Linux 内核的时候就会使用新的配置文件，默认就会使能 LAN8720A 的驱动。

2、通过图形界面保存配置文件

相比于第 1 种直接另存为.config 文件，第 2 种方法就很“文雅”了，在图形界面中保存配置文件，在图形界面中会有“< Save >”选项，如图 37.4.4.1 所示：

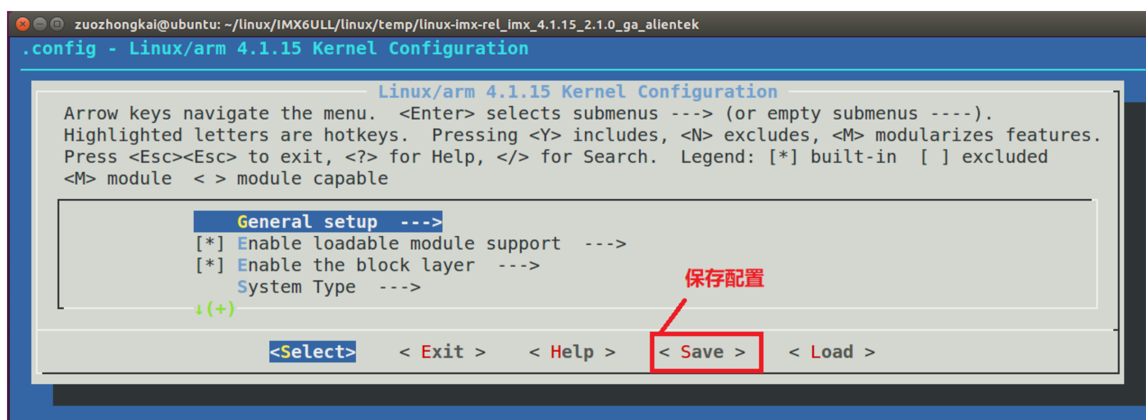


图 37.4.4.1 保存配置

通过键盘的“→”键，移动到“< Save >”选项，然后按下回车键，打开文件名输入对话框，如图 37.4.4.2 所示：

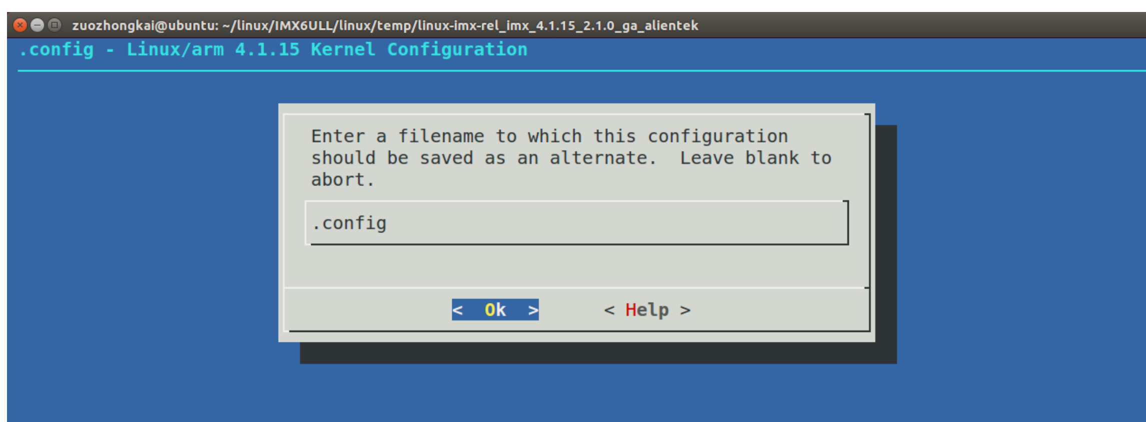


图 37.4.4.2 输入文件名

在图 37.4.4.2 中输入要保存的文件名，可以带路径，一般是相对路径(相对于 Linux 内核源码根目录)。比如我们要将新的配置文件保存到目录 arch/arm/configs 下，文件名为 imx_alientek_emmc_defconfig，也就是用新的配置文件替换掉老的默认配置文件。那么我们在图 37.4.4.2 中输入“arch/arm/configs/imx_alientek_emmc_defconfig”即可，如图 37.4.4.3 所示：

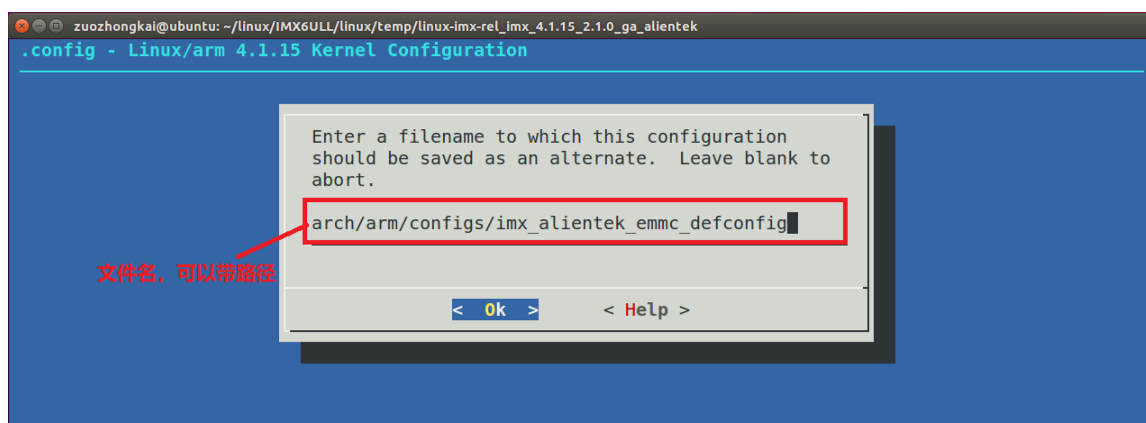


图 37.4.4.3 输入文件名

设置好文件名以后选择下方的“< Ok >”按钮，保存文件并退出。退出以后再打开 imx_alientek_emmc_defconfig 文件，就会在此文件中找到“CONFIG_SMSC_PHY=y”这一行，如图 37.4.4.4 所示：

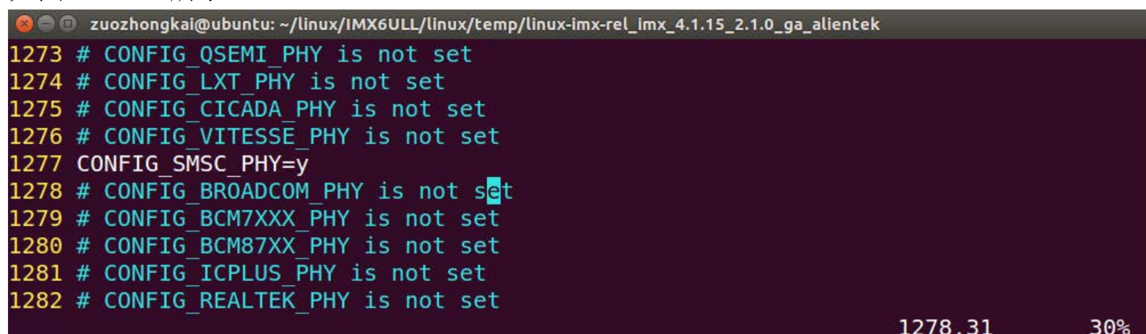


图 37.4.4.4 新的配置文件

同样的，使用“make imx_alientek_emmc_defconfig”重新配置 Linux 内核的时候，LAN8720A 的驱动就会使能，并被编译进 Linux 镜像文件 zImage 中。

关于 Linux 内核的移植就讲解到这里，简单总结一下移植步骤：

①、在 Linux 内核中查找可以参考的板子，一般都是半导体厂商自己做的开发板。

②、编译出参考板子对应的 zImage 和.dtb 文件。

③、使用参考板子的 zImage 文件和.dtb 文件在我们所使用的板子上启动 Linux 内核，看能否启动。

④、如果能启动的话就万事大吉，如果不能启动那就悲剧了，需要调试 Linux 内核。不过一般都会参考半导体官方的开发板设计自己的硬件，所以大部分情况下都会启动起来。启动 Linux 内核用到的外设不多，一般就 DRAM(Uboot 都初始化好的)和串口。作为终端使用的串口一般都会参考半导体厂商的 Demo 板。

⑤、修改相应的驱动，像 NAND Flash、EMMC、SD 卡等驱动官方的 Linux 内核都是已经提供好了，基本不会出问题。重点是网络驱动，因为 Linux 驱动开发一般都要通过网络调试代码，所以一定要确保网络驱动工作正常。如果是处理器内部 MAC+外部 PHY 这种网络方案的话，一般网络驱动都很好处理，因为在 Linux 内核中是有外部 PHY 通用驱动的。只要设置好复位引脚、PHY 地址信息基本上都可以驱动起来。

⑥、Linux 内核启动以后需要根文件系统，如果没有根文件系统的话肯定会崩溃，所以确定 Linux 内核移植成功以后就要开始根文件系统的构建。

第三十八章 根文件系统构建

Linux“三巨头”已经完成了 2 个了，就剩最后一个 rootfs(根文件系统)了，本章我们就来学习一下根文件系统的组成以及如何构建根文件系统。这是 Linux 移植的最后一步，根文件系统构建好以后就意味着我们已经拥有了一个完整的、可以运行的最小系统。以后我们就在这个最小系统上编写、测试 Linux 驱动，移植一些第三方组件，逐步的完善这个最小系统。最终得到一个功能完善、驱动齐全、相对完善的操作系统。

38.1 根文件系统简介

根文件系统一般也叫做 rootfs，那么什么叫根文件系统？看到“文件系统”这四个字，很多人，包括我第一反应就是 FATFS、FAT、EXT4、YAFFS 和 NTFS 等这样的文件系统。在这里，根文件系统并不是 FATFS 这样的文件系统代码，EXT4 这样的文件系统代码属于 Linux 内核的一部分。Linux 中的根文件系统更像是一个文件夹或者叫做目录(在我看来就是一个文件夹，只不过是特殊的文件夹)，在这个目录里面会有很多的子目录。根目录下和子目录中会有很多的文件，这些文件是 Linux 运行所必须的，比如库、常用的软件和命令、设备文件、配置文件等等。以后我们说到文件系统，如果不特别指明，统一表示根文件系统。对于根文件系统专业的解释，百度百科上是这么说的(原谅我把百度百科引用为专业解释，因为我实在找不到根文件系统的最初定义，也不要建议我到哪些 404 网站去查找，毕竟我胖，我怕翻到一般梯子不稳把我摔丑了)：

根文件系统首先是内核启动时所 mount(挂载)的第一个文件系统，内核代码映像文件保存在根文件系统中，而系统引导启动程序会在根文件系统挂载之后从中把一些基本的初始化脚本和服务等加载到内存中去运行。

百度百科上说内核代码镜像文件保存在根文件系统中，但是我们嵌入式 Linux 并没有将内核代码镜像保存在根文件系统中，而是保存到了其他地方。比如 NAND Flash 的指定存储地址、EMMC 专用分区中。根文件系统是 Linux 内核启动以后挂载(mount)的第一个文件系统，然后从根文件系统中读取初始化脚本，比如 rcS，inittab 等。根文件系统和 Linux 内核是分开的，单独的 Linux 内核是没法正常工作的，必须要搭配根文件系统。如果不提供根文件系统，Linux 内核在启动的时候就会提示内核崩溃(Kernel panic)的提示，这个在 37.2.4 小节已经说过了。

根文件系统的这个“根”字就说明了这个文件系统的重要性，它是其他文件系统的根，没有这个“根”，其他的文件系统或者软件就别想工作。比如我们常用的 ls、mv、ifconfig 等命令其实就是一个个小软件，只是这些软件没有图形界面，而且需要输入命令来运行。这些小软件就保存在根文件系统中，这些小软件是怎么来的呢？这个就是我们本章教程的目的，教大家来构建自己的根文件系统，这个根文件系统是满足 Linux 运行的最小根文件系统，后续我们可以根据自己的实际工作需求不断的去填充这个最小根文件系统，最终使其成为一个相对完善的根文件系统。

在构建根文件系统之前，我们先来看一下根文件系统里面大概都有些什么内容，以 Ubuntu 为例，根文件系统的目录名字为‘/’，没看错就是一个斜杠，所以输入如下命令就可以进入根目录中：

```
cd /            //进入根目录
```

进入根目录以后输入“ls”命令查看根目录下的内容都有哪些，结果如图 38.1.1 所示：

```
zuozhongkai@ubuntu:/$ cd /
zuozhongkai@ubuntu:/$ ls
bin          build-trusted  home            lib32       media  root  srv  var
boot         cdrom          initrd.img      lib64       mnt    run   sys  vmlinuz
build-basic  dev            initrd.img.old  libx32      opt    sbin  tmp  vmlinuz.old
build-optee  etc            lib             lost+found  proc   snap  usr
zuozhongkai@ubuntu:/$
```

图 38.1.1 Ubuntu 根目录

图 38.1.1 中根目录下子目录和文件不少，但是这些都是 Ubuntu 所需要的，其中有很多子目录和文件我们嵌入式 Linux 是用不到的，所以这里就讲解一些常用的子目录：

1、/bin 目录

看到“bin”大家应该能想到 bin 文件，bin 文件就是可执行文件。所以此目录下存放着系统需要的可执行文件，一般都是一些命令，比如 ls、mv 等命令。此目录下的命令所有的客户都可以使用。

2、/dev 目录

dev 是 device 的缩写，所以此目录下的文件都是和设备有关的，此目录下的文件都是设备文件。在 Linux 下一切皆文件，即使是硬件设备，也是以文件的形式存在的，比如/dev/ttymxc0(I.MX6ULL 根目录会有此文件)就表示 I.MX6ULL 的串口 0，我们要想通过串口 0 发送或者接收数据就要操作文件/dev/ttymxc0，通过对文件/dev/ttymxc0 的读写操作来实现串口 0 的数据收发。

3、/etc 目录

此目录下存放着各种配置文件，大家可以进入 Ubuntu 的 etc 目录看一下，里面的配置文件非常多！但是在嵌入式 Linux 下此目录会很简洁。

4、/lib 目录

lib 是 library 的简称，也就是库的意思，因此此目录下存放着 Linux 所必须的库文件。这些库文件是共享库，命令和用户编写的应用程序要使用这些库文件。

5、/mnt 目录

临时挂载目录，一般是空目录，可以在此目录下创建空的子目录，比如/mnt/sd、/mnt/usb，这样就可以将 SD 卡或者 U 盘挂载到/mnt/sd 或者/mnt/usb 目录中。

6、/proc 目录

此目录一般是空的，当 Linux 系统启动以后会将此目录作为 proc 文件系统的挂载点，proc 是个虚拟文件系统，没有实际的存储设备。proc 里面的文件都是临时存在的，一般用来存储系统运行信息文件。

7、/usr 目录

要注意，usr 不是 user 的缩写，而是 Unix Software Resource 的缩写，也就是 Unix 操作系统软件资源目录。这里有个小知识点，那就是 Linux 一般被成为类 Unix 操作系统，苹果的 MacOS 也是类 Unix 操作系统。关于 Linux 和 Unix 操作系统的渊源大家可以直接在网上找 Linux 的发展历史来看。既然是软件资源目录，因此/usr 目录下也存放着很多软件，一般系统安装完成以后此目录占用的空间最多。

8、/var 目录

此目录存放一些可以改变的数据。

9、/sbin 目录

此目录页用户存放一些可执行文件，但是此目录下的文件或者说命令只有管理员才能使用，主要用户系统管理。

10、/sys 目录

系统启动以后此目录作为 sysfs 文件系统的挂载点，sysfs 是一个类似于 proc 文件系统的特殊文件系统，sysfs 也是基于 ram 的文件系统，也就是说它也没有实际的存储设备。此目录是系统设备管理的重要目录，此目录通过一定的组织结构向用户提供详细的内核数据结构信息。

11、/opt

可选的文件、软件存放区，由用户选择将哪些文件或软件放到此目录中。

关于 Linux 的根目录就介绍到这里，接下来的构建根文件系统就是研究如何创建上面这些子目录以及子目录中的文件。

38.2 BusyBox 构建根文件系统

38.2.1 BusyBox 简介

上一小节说了，根文件系统里面就是一堆的可执行文件和其他文件组成的？难道我们得一个一个的从网上去下载这些文件？显然这是不现实的！那么有没有人或者组织专门干这个事呢？他们负责“收集”这些文件，然后将其打包，像我们这样的开发者可以直接拿来用。答案是有的，它就叫做 BusyBox！其名字分为“Busy”和“Box”，也就是忙碌的盒子。盒子是用来放东西的，忙碌的是因为它要提供根文件系统所需的文件，所以忙碌。BusyBox 是一个集成了大量的 Linux 命令和工具的软件，像 ls、mv、ifconfig 等命令 BusyBox 都会提供。BusyBox 就是一个大的工具箱，这个工具箱里面集成了 Linux 的许多工具和命令。一般下载 BusyBox 的源码，然后配置 BusyBox，选择自己想要的功能，最后编译即可。

BusyBox 可以在其官网下载到，官网地址为：https://busybox.net/，官网比较简陋，如图 38.2.1.1 所示：

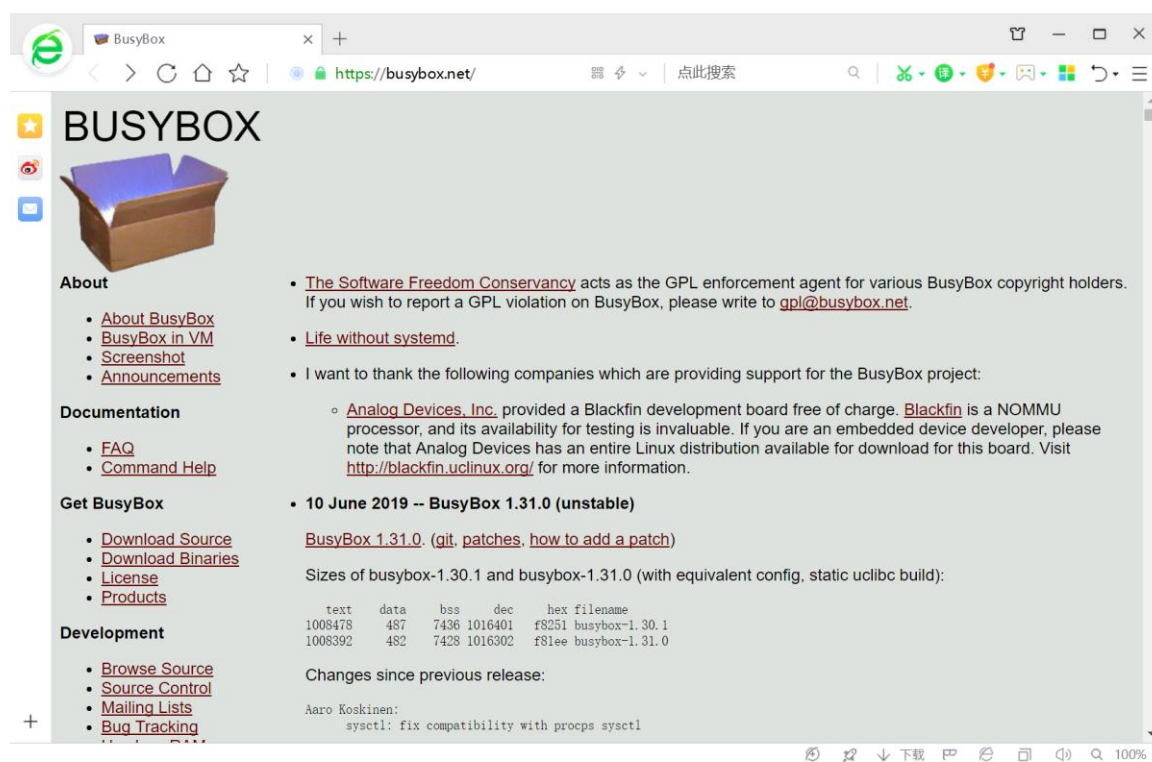


图 38.2.1.1 BusyBox 官网

在官网左侧的“Get BusyBox”栏有一行“Download Source”，点击“Download Source”即可打开 BusyBox 的下载页，如图 38.2.1.2 所示：

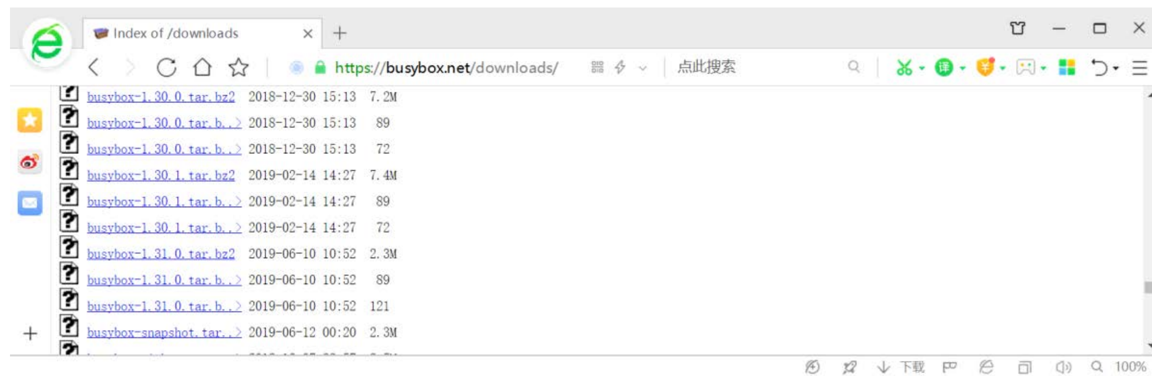


图 38.2.1.2 BusyBox 下载页

从图 38.2.1.2 可以看出，目前最新的 BusyBox 版本是 1.31.0，不过我建议大家使用我们开发板光盘里面提供的 1.29.0 版本的 BusyBox。因为笔者测试 1.29.0 版本目前还没有出现任何问题，路径为：1、例程源码->6、BusyBox 源码->busybox-1.29.0.tar.bz2，BusyBox 准备好以后就可以构建根文件系统了。

38.2.2 编译 BusyBox 构建根文件系统

一般我们在 Linux 驱动开发的时候都是通过 nfs 挂载根文件系统的，当产品最终上市开卖的时候才会将根文件系统烧写到 EMMC 或者 NAND 中。所以要在 4.2.1 小节中设置的 nfs 服务器目录中创建一个名为 rootfs 的子目录(名字大家可以随意起，为了方便就用了 rootfs)，比如我的电脑中“/home/zuozhongkai/linux/nfs”就是我设置的 NFS 服务器目录，使用如下命令创建名为 rootfs 的子目录：

```
mkdir rootfs
```

创建好的 rootfs 子目录就用来存放我们的根文件系统了。

将 busybox-1.29.0.tar.bz2 发送到 Ubuntu 中，存放位置大家随便选择。然后使用如下命令将其解压：

```
tar -vxjf busybox-1.29.0.tar.bz2
```

解压完成以后进入到 busybox-1.29.0 目录中，此目录中的文件和文件夹如图 38.2.2.1 所示：

```
zuozhongkai@ubuntu:~/linux/busybox$ cd busybox-1.29.0/
zuozhongkai@ubuntu:~/linux/busybox/busybox-1.29.0$ ls
applets        coreutils    init         Makefile                 NOFORK_NOEXEC.lst       selinux
applets_sh     debianutils  INSTALL      Makefile.custom          NOFORK_NOEXEC.sh        shell
arch           docs         klibc-utils  Makefile.flags           printutils              size_single_applets.sh
archival       e2fsprogs    libbb        Makefile.help            procps                  sysklogd
AUTHORS        editors      libpwdgrp    make_single_applets.sh   qemu_multiarch_testing  testsuite
Config.in      examples     LICENSE      miscutils                README                  TODO
configs        findutils    loginutils   modutils                 runit                   TODO_unicode
console-tools  include      mailutils    networking               scripts                 util-linux
zuozhongkai@ubuntu:~/linux/busybox/busybox-1.29.0$
```

图 38.2.2.1 busybox-1.29.0 目录内容

1、修改 Makefile，添加编译器

同 Uboot 和 Linux 移植一样，打开 busybox 的顶层 Makefile，添加 ARCH 和 CROSS_COMPILE 的值，如下所示：

示例代码 38.2.2.1 Makefile 代码段

```
164 CROSS_COMPILE ?= /usr/local/arm/gcc-linaro-4.9.4-2017.01-
x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
......
190 ARCH ?= arm
```

在示例代码 38.2.2.1 中 CORSS_COMPILE 使用了绝对路径！主要是为了防止编译出错。

2、busybox 中文字符支持

如果默认直接编译 busybox 的话，在使用 SecureCRT 的时候中文字符是显示不正常的，中文字符会显示为“?”，比如你的中文目录，中文文件都显示为“?”。不知道从哪个版本开始 busybox 中的 shell 命令对中文输入即显示做了限制，即使内核支持中文但在 shell 下也依然无法正确显示。

所以我们需要修改 busybox 源码，取消 busybox 对中文显示的限制，打开文件 busybox-1.29.0/libbb/printable_string.c，找到函数 printable_string，缩减后的函数内容如下：

示例代码 38.2.2.2 libbb/printable_string.c 代码段

```
12 const char* FAST_FUNC printable_string(uni_stat_t *stats, const char
*str)
13 {
14  char *dst;
15  const char *s;
16
17  s = str;
18  while (1) {
19      unsigned char c = *s;
20      if (c == '\0') {
......
28      }
29      if (c < ' ')
30          break;
31      if (c >= 0x7f)
32          break;
33      s++;
34  }
35
36 #if ENABLE_UNICODE_SUPPORT
37  dst = unicode_conv_to_printable(stats, str);
38 #else
39  {
40      char *d = dst = xstrdup(str);
41      while (1) {
42          unsigned char c = *d;
43          if (c == '\0')
44              break;
45          if (c < ' ' || c >= 0x7f)
46              *d = '?';
47          d++;
48      }
......
55 #endif
```

```
56 return auto_string(dst);
57 }
```

第 31 和 32 行，当字符大于 0X7F 以后就跳出去了。

第 45 和 46 行，如果支持 UNICODE 码的话，当字符大于 0X7F 就直接输出‘?’。

所以我们需要对这 4 行代码进行修改，修改以后如下所示：

示例代码 38.2.2.3 libbb/printable_string.c 代码段

```
12 const char* FAST_FUNC printable_string(uni_stat_t *stats, const char
*str)
13 {
14  char *dst;
15  const char *s;
16
17  s = str;
18  while (1) {
......
30      if (c < ' ')
31          break;
32      /* 注释掉下面这个两行代码 */
33      /* if (c >= 0x7f)
34          break; */
35      s++;
36  }
37
38 #if ENABLE_UNICODE_SUPPORT
39  dst = unicode_conv_to_printable(stats, str);
40 #else
41  {
42      char *d = dst = xstrdup(str);
43      while (1) {
44          unsigned char c = *d;
45          if (c == '\0')
46              break;
47          /* 修改下面代码 */
48          /* if (c < ' ' || c >= 0x7f) */
49          if( c < ' ')
50              *d = '?';
51          d++;
52      }
......
59 #endif
60  return auto_string(dst);
61 }
```

示例代码 38.2.2.3 中红色部分的代码就是被修改以后的，主要就是禁止字符大于 0X7F 以后 break 和输出‘?’。

接着打开文件 busybox-1.29.0/libbb/unicode.c，找到如下内容：

示例代码 38.2.2.4 libbb/unicode.c 代码段

```
1003 static char* FAST_FUNC unicode_conv_to_printable2(uni_stat_t
*stats, const char *src, unsigned width, int flags)
1004 {
1005    char *dst;
1006    unsigned dst_len;
1007    unsigned uni_count;
1008    unsigned uni_width;
1009
1010    if (unicode_status != UNICODE_ON) {
1011        char *d;
1012        if (flags & UNI_FLAG_PAD) {
1013            d = dst = xmalloc(width + 1);
......
1022                *d++ = (c >= ' ' && c < 0x7f) ? c : '?';
1023                src++;
1024            }
1025            *d = '\0';
1026        } else {
1027            d = dst = xstrndup(src, width);
1028            while (*d) {
1029                unsigned char c = *d;
1030                if (c < ' ' || c >= 0x7f)
1031                    *d = '?';
1032                d++;
1033            }
1034        }
......
1040        return dst;
1041    }
......
1130
1131    return dst;
1132 }
```

第 1022 行，当字符大于 0X7F 以后，*d++就为‘?’。

第 1030 和 1031 行，当字符大于 0X7F 以后，*d 也为‘?’。

修改示例代码 38.2.2.4，修改后内容如下所示：

示例代码 38.2.2.5 libbb/unicode.c 代码段

```
1003 static char* FAST_FUNC unicode_conv_to_printable2(uni_stat_t
*stats, const char *src, unsigned width, int flags)
```

```
1004 {
1005     char *dst;
1006     unsigned dst_len;
1007     unsigned uni_count;
1008     unsigned uni_width;
1009 
1010     if (unicode_status != UNICODE_ON) {
1011         char *d;
1012         if (flags & UNI_FLAG_PAD) {
1013             d = dst = xmalloc(width + 1);
......
1022                 /* 修改下面一行代码 */
1023                 /* *d++ = (c >= ' ' && c < 0x7f) ? c : '?'; */
1024                 *d++ = (c >= ' ') ? c : '?';
1025                 src++;
1026             }
1027             *d = '\0';
1028         } else {
1029             d = dst = xstrndup(src, width);
1030             while (*d) {
1031                 unsigned char c = *d;
1032                 /* 修改下面一行代码 */
1033                 /* if (c < ' ' || c >= 0x7f) */
1034                 if(c < ' ')
1035                     *d = '?';
1036                 d++;
1037             }
1038         }
......
1044         return dst;
1045     }
......
1047 
1048     return dst;
1049 }
```

示例代码 38.2.2.5 中红色部分的代码就是被修改以后的，同样主要是禁止字符大于 0X7F 的时候设置为‘?’。busybox 中文字符支持跟代码修改有关的就改好了，最后还需要配置 busybox 来使能 unicode 码，这个稍后我们配置 busybox 的时候在设置。

3、配置 busybox

根我们编译 Uboot、Linux kernel 一样，我们要先对 busybox 进行默认的配置，有以下几种配置选项：

①、defconfig，缺省配置，也就是默认配置选项。

②、allyesconfig，全选配置，也就是选中 busybox 的所有功能。

③、allnoconfig，最小配置。

我们一般使用默认配置即可，因此使用如下命令先使用默认配置来配置一下 busybox：

```
make defconfig
```

busybox 也支持图形化配置，通过图形化配置我们可以进一步选择自己想要的功能，输入如下命令打开图形化配置界面：

```
make menuconfig
```

打开以后如图 38.2.2.2 所示：

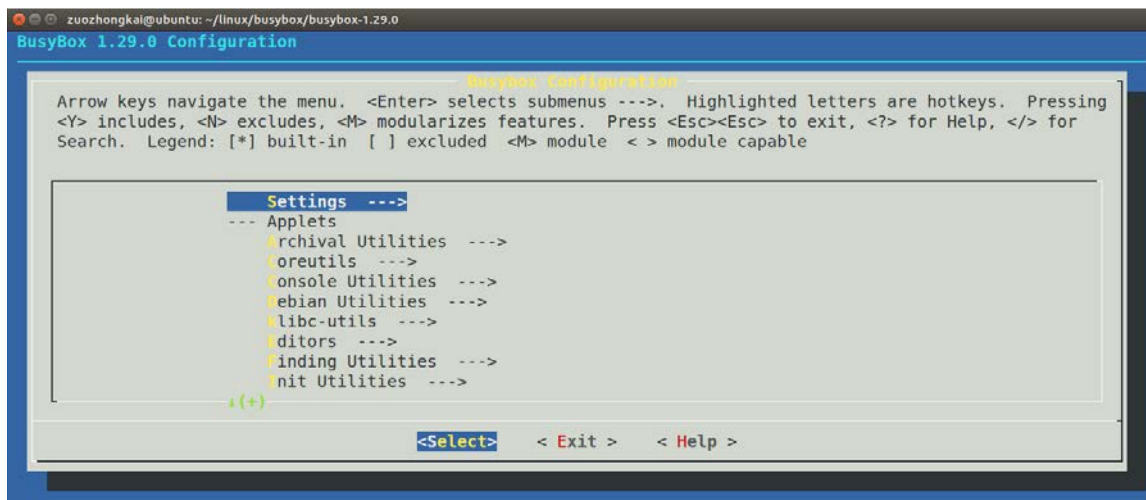


图 38.2.2.2 busybox 图形化配置界面

配置路径如下：

```
Location:
    -> Settings
        -> Build static binary (no shared libs)
```

选项“Build static binary (no shared libs)”用来决定是静态编译 busybox 还是动态编译，静态编译的话就不需要库文件，但是编译出来的库会很大。动态编译的话要求根文件系统中有库文件，但是编译出来的 busybox 会小很多。这里我们不能采用静态编译！因为采用静态编译的话 DNS 会出问题！无法进行域名解析，配置如图 38.2.2.3 所示：

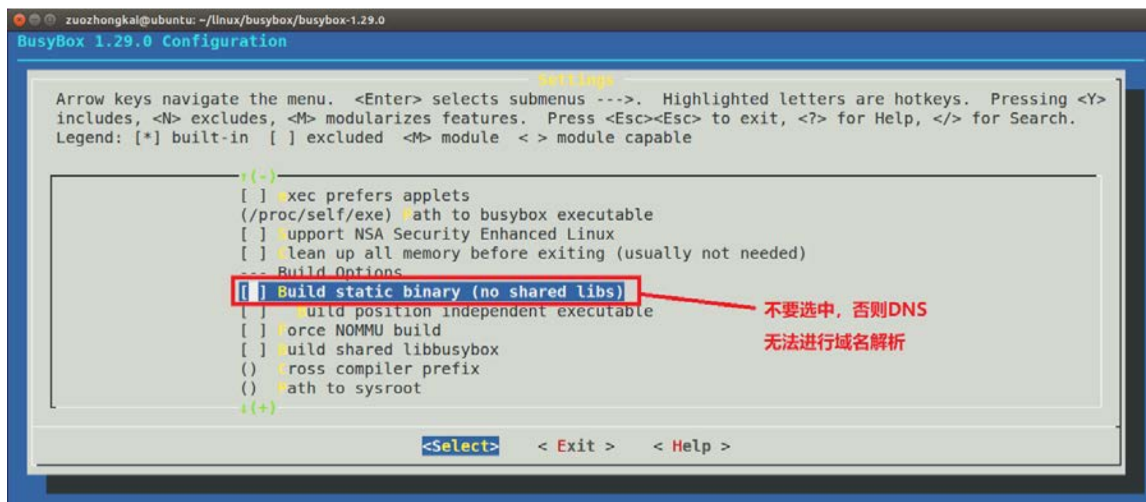


图 38.2.2.3 不选择“Build static binary (no shared libs)”

继续配置如下路径配置项：

Location:
-> Settings
-> vi-style line editing commands

结果如图 38.2.2.4 所示：

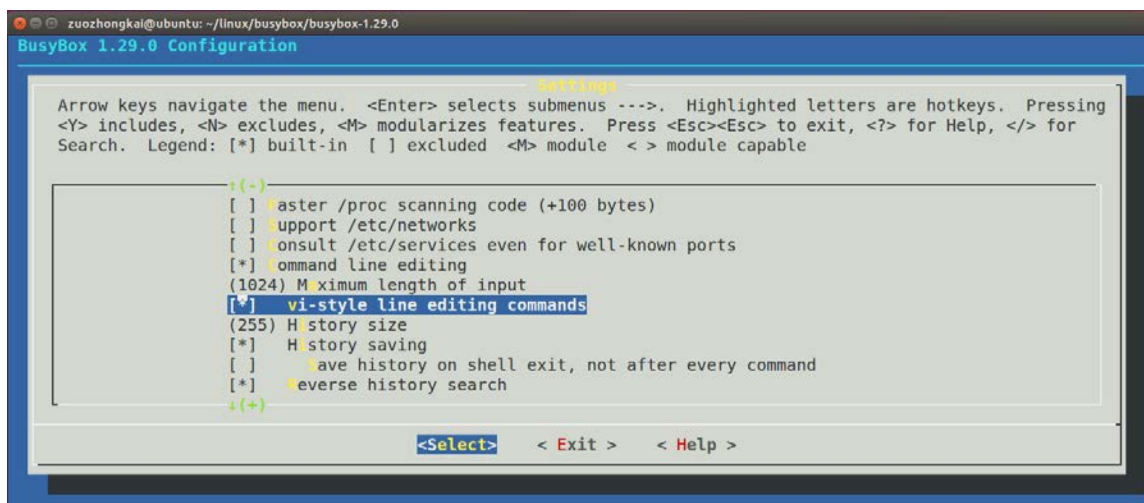


图 38.2.2.4 选择“vi-style line editing commands ”

继续配置如下路径配置项：

Location:
-> Linux Module Utilities
-> Simplified modutils

默认会选中“Simplified modutils”，这里我们要取消勾选！！结果如图 38.2.2.5 所示：

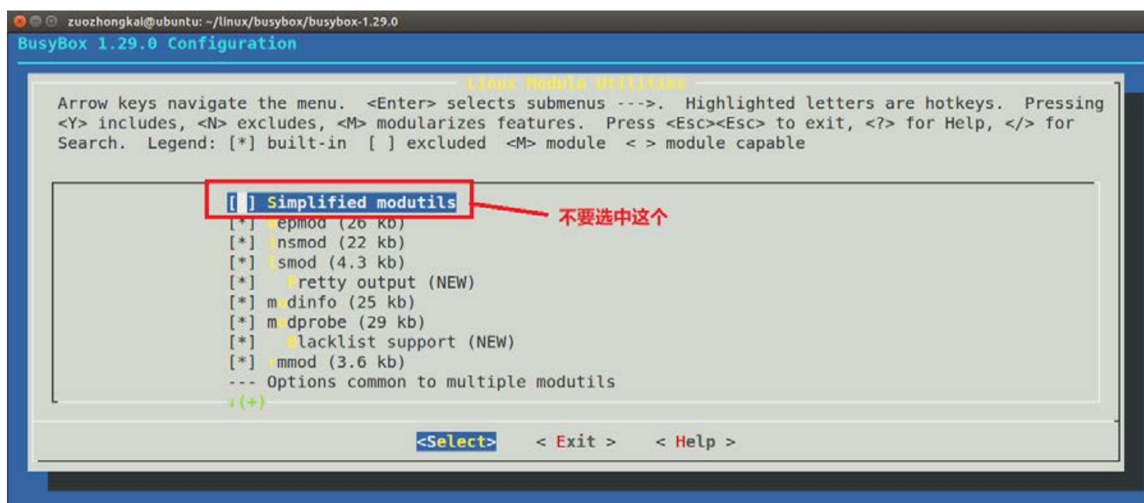


图 38.2.2.5 取消选中“Simplified modutils”

继续配置如下路径配置项：

Location:
-> Linux System Utilities
-> mdev (16 kb) //确保下面的全部选中，默认都是选中的

结果如图 38.2.2.6 所示：

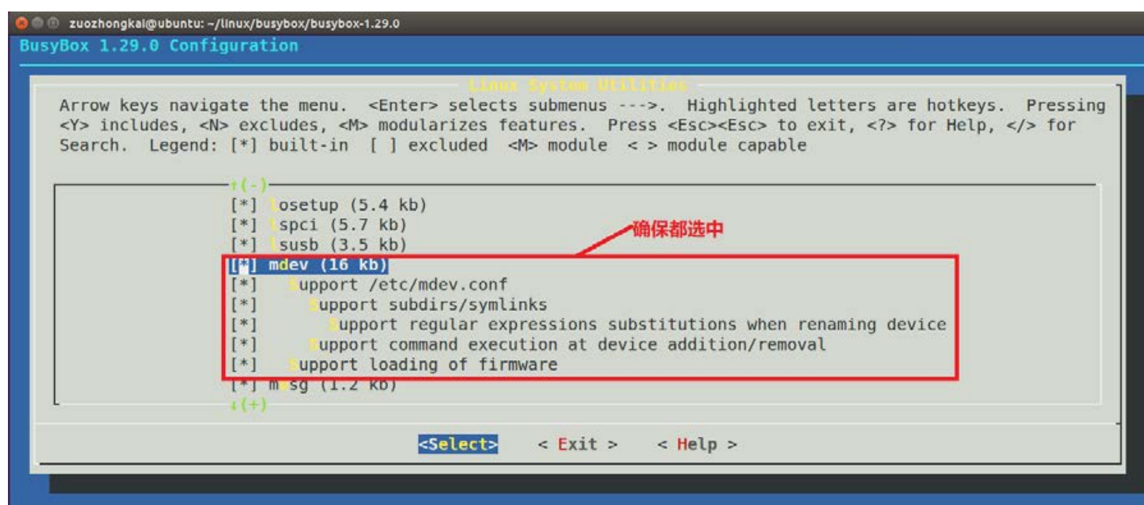


图 38.2.2.6 “mdev”配置项

最后就是使能 busybox 的 unicode 编码以支持中文，配置路径如下：

```
Location:
    -> Settings
        -> Support Unicode                                                  //选中
            -> Check $LC_ALL, $LC_CTYPE and $LANG environment variables    //选中
```

结果如图 38.2.2.7 所示：

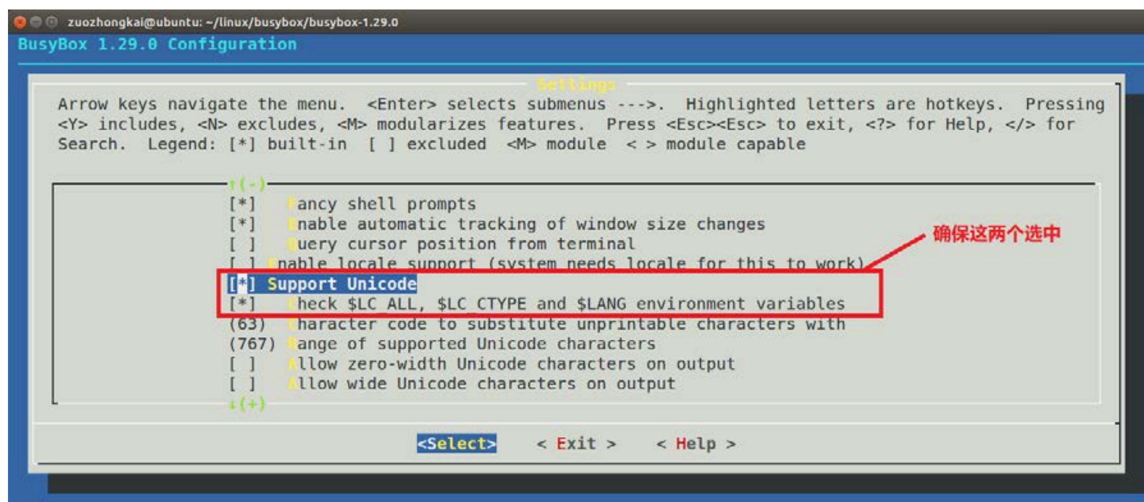


图 38.2.2.7 中文支持

busybox 的配置就到此结束了，大家也可以根据自己的实际需求选择配置其他的选项，不过对于初学者笔者不建议再做其他的修改，可能会出现编译出错的情况发生。

4、编译 busybox

配置好 busybox 以后就可以编译了，我们可以指定编译结果的存放目录，我们肯定要将编译结果存放到前面创建的 rootfs 目录中，输入如下命令：

```
make install CONFIG_PREFIX=/home/zuozhongkai/linux/nfs/rootfs
```

COFIG_PREFIX 指定编译结果的存放目录，比如我存放到“/home/zuozhongkai/linux/nfs/rootfs”目录中，等待编译完成。编译完成以后如图 38.2.2.8 所示：

```
./_install//usr/sbin/ubirename -> ../../bin/busybox
./_install//usr/sbin/ubirmvol -> ../../bin/busybox
./_install//usr/sbin/ubirsvol -> ../../bin/busybox
./_install//usr/sbin/ubiupdatevol -> ../../bin/busybox
./_install//usr/sbin/udhcpd -> ../../bin/busybox

--------------------------------------------------
You will probably need to make your busybox binary
setuid root to ensure all configured applets will
work properly.
--------------------------------------------------

zuozhongkai@ubuntu:~/linux/busybox/busybox-1.29.0$
```

图 38.2.2.8 busybox 编译完成

编译完成以后会在 busybox 的所有工具和文件就会被安装到 rootfs 目录中，rootfs 目录内容如图 38.2.2.9 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs$ ls
bin  linuxrc  sbin  usr
zuozhongkai@ubuntu:~/linux/nfs/rootfs$
```

图 38.2.2.9 rootfs 目录

从图 38.2.2.9 可以看出，rootfs 目录下有 bin、sbin 和 usr 这三个目录，以及 linuxrc 这个文件。前面说过 Linux 内核 init 进程最后会查找用户空间的 init 程序，找到以后就会运行这个用户空间的 init 程序，从而切换到用户态。如果 bootargs 设置 init=/linuxrc，那么 linuxrc 就是可以作为用户空间的 init 程序，所以用户态空间的 init 程序是 busybox 来生成的。

busybox 的工作就完成了，但是此时的根文件系统还不能使用，还学要一些其他的文件，所以我们继续来完善 rootfs。

38.2.3 向根文件系统添加 lib 库

1、向 rootfs 的“/lib”目录添加库文件

Linux 中的应用程序一般都是需要动态库的，当然你也可以编译成静态的，但是静态的可执行文件会很大。如果编译为动态的话就需要动态库，所以我们需要先根文件系统中添加动态库。在 rootfs 中创建一个名为“lib”的文件夹，命令如下：

```
mkdir lib
```

lib 文件创建好了，库文件从哪里来呢？lib 库文件从交叉编译器中获取，前面我们搭建交叉编译环境的时候将交叉编译器存放到了“/usr/local/arm/”目录中。交叉编译器里面有很多的库文件，这些库文件具体是做什么的我们作为初学者肯定不知道，既然我不知道那就简单粗暴的把所有的库文件都放到我们的根文件系统中。这样做出来的根文件系统肯定很大，但是我们现在是学习阶段，还做不了裁剪。这就是为什么我们推荐大家购买 512MB+8GB 版本的 EMMC 核心版，如果后面要学习 QT 的话那占用的空间将更大，不裁剪的话 512MB 的 NAND 完全不够用的！而裁剪又是需要经验的，我们都是初学者，哪里来的经验啊。所以我们推荐初学者购买 EMMC 版核心板并不是说为了多赚大家的钱，而是从实际角度考虑的。

进入如下路径对应的目录：

```
/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/libc/lib
```

此目录下有很多的*so*(*是通配符)和.a 文件，这些就是库文件，将此目录下所有的*so*和.a 文件都拷贝到 rootfs/lib 目录中，拷贝命令如下：

```
cp *so* *.a /home/zuozhongkai/linux/nfs/rootfs/lib/ -d
```

后面的“-d”表示拷贝符号链接，这里有个比较特殊的库文件：ld-linux-armhf.so.3，此库文件也

是个符号链接，相当于 Windows 下的快捷方式。会链接到库 ld-2.19-2014.08-1-git.so 上，输入命令“ls ld-linux-armhf.so.3 -l”查看此文件详细信息，如图 38.2.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs$ cd lib/
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$ ls ld-linux-armhf.so.3 -l
lrwxrwxrwx 1 zuozhongkai zuozhongkai 24 Jun 13 12:36 ld-linux-armhf.so.3 -> ld-2.19-2014.08-1-git.so
```

图 38.2.3.1 文件 ld-linux-armhf.so.3

从图 38.2.3.1 可以看出，ld-linux-armhf.so.3 后面有个“->”，表示其是个软连接文件，链接到文件 ld-2.19-2014.08-1-git.so，因为其是一个“快捷方式”，因此大小只有 24B。但是，ld-linux-armhf.so.3 不能作为符号链接，否则的话在根文件系统中执行程序无法执行！所以我们需要 ld-linux-armhf.so.3 完成逆袭，由“快捷方式”变为“本尊”，方法很简单，那就是重新复制 ld-linux-armhf.so.3，只是不复制软链接即可，先将 rotfs/lib 中的 ld-linux-armhf.so.3 文件删除掉，命令如下：

```
rm ld-linux-armhf.so.3
```

然后重新进入到 /usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/libc/lib 目录中，重新拷贝 ld-linux-armhf.so.3，命令如下：

```
cp ld-linux-armhf.so.3 /home/zuozhongkai/linux/nfs/rootfs/lib/
```

拷贝完成以后再到 rootfs/lib 目录下查看 ld-linux-armhf.so.3 文件详细信息，如图 38.2.3.2 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$ rm  ld-linux-armhf.so.3
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$ ls ld-linux-armhf.so.3 -l
-rwxr-xr-x 1 zuozhongkai zuozhongkai 724392 Jun 13 12:59 ld-linux-armhf.so.3
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$
```

图 38.2.3.2 文件 ld-linux-armhf.so.3

从图 38.2.3.2 可以看出，此时 ld-linux-armhf.so.3 已经不是软连接了，而是实实在在的一个库文件，而且文件大小为 724392B。

继续进入如下目录中：

```
/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/lib
```

此目录下也有很多的的*so*和.a 库文件，我们将其也拷贝到 rootfs/lib 目录中，命令如下：

```
cp *so* *.a /home/zuozhongkai/linux/nfs/rootfs/lib/ -d
```

rootfs/lib 目录的库文件就这些了，完成以后的 rootfs/lib 目录如图 38.2.3.3 所示：

```
zuozhongkai@ubuntu: ~/linux/nfs/rootfs/lib
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$ ls
ld-2.19-2014.08-1-git.so            libgomp.so                          libpthread-2.19-2014.08-1-git.so
ld-linux-armhf.so.3                 libgomp.so.1                        libpthread.so.0
libanl-2.19-2014.08-1-git.so        libgomp.so.1.0.0                    libresolv-2.19-2014.08-1-git.so
libanl.so.1                         libitm.a                            libresolv.so.2
libasan.a                           libitm.so                           librt-2.19-2014.08-1-git.so
libasan.so                          libitm.so.1                         librt.so.1
libasan.so.1                        libitm.so.1.0.0                     libSegFault.so
libasan.so.1.0.0                    libm-2.19-2014.08-1-git.so          libssp.a
libatomic.a                         libmemusage.so                      libssp_nonshared.a
libatomic.so                        libm.so.6                           libssp.so
libatomic.so.1                      libnsl-2.19-2014.08-1-git.so        libssp.so.0
libatomic.so.1.1.0                  libnsl.so.1                         libssp.so.0.0.0
libBrokenLocale-2.19-2014.08-1-git.so  libnss_compat-2.19-2014.08-1-git.so  libstdc++.a
libBrokenLocale.so.1                libnss_compat.so.2                  libstdc++.so
libc-2.19-2014.08-1-git.so          libnss_db-2.19-2014.08-1-git.so     libstdc++.so.6
libcrypt-2.19-2014.08-1-git.so      libnss_db.so.2                      libstdc++.so.6.0.20
libcrypt.so.1                       libnss_dns-2.19-2014.08-1-git.so    libstdc++.so.6.0.20-gdb.py
libc.so.6                           libnss_dns.so.2                     libsupc++.a
libdl-2.19-2014.08-1-git.so         libnss_files-2.19-2014.08-1-git.so  libthread_db-1.0.so
libdl.so.2                          libnss_files.so.2                   libthread_db.so.1
libgcc_s.so                         libnss_hesiod-2.19-2014.08-1-git.so libubsan.a
libgcc_s.so.1                       libnss_hesiod.so.2                  libubsan.so
libgfortran.a                       libnss_nis-2.19-2014.08-1-git.so    libubsan.so.0
libgfortran.so                      libnss_nisplus-2.19-2014.08-1-git.so libubsan.so.0.0.0
libgfortran.so.3                    libnss_nisplus.so.2                 libutil-2.19-2014.08-1-git.so
libgfortran.so.3.0.0                libnss_nis.so.2                     libutil.so.1
libgomp.a                           libpcprofile.so
zuozhongkai@ubuntu:~/linux/nfs/rootfs/lib$
```

图 38.2.3.3 lib 目录

2、向 rootfs 的“usr/lib”目录添加库文件

在 rootfs 的 usr 目录下创建一个名为 lib 的目录，将如下目录中的库文件拷贝到 rootfs/usr/lib 目录下：

```
/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf/arm-linux-gnueabihf/libc/usr/lib
```

将此目录下的 so 和.a 库文件都拷贝到 rootfs/usr/lib 目录中，命令如下：

```
cp *so* *.a /home/zuozhongkai/linux/nfs/rootfs/usr/lib/ -d
```

完成以后的 rootfs/usr/lib 目录如图 38.2.3.4 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs/usr/lib$ ls
libanl.a                libcrypt_pic.a  libnsl.a            libnss_nis_pic.a        librpcsvc_p.a
libanl_p.a              libcrypt.so     libnsl_p.a          libnss_nisplus_pic.a    librt.a
libanl_pic.a            libc.so         libnsl_pic.a        libnss_nisplus.so       librt_p.a
libanl.so               libdl.a         libnsl.so           libnss_nis.so           librt_pic.a
libBrokenLocale.a       libdl_p.a       libnss_compat_pic.a libpthread.a            librt.so
libBrokenLocale_p.a     libdl_pic.a     libnss_compat.so    libpthread_nonshared.a  libthread_db_pic.a
libBrokenLocale_pic.a   libdl.so        libnss_db_pic.a     libpthread_p.a          libthread_db.so
libBrokenLocale.so      libg.a          libnss_db.so        libpthread.so           libutil.a
libc.a                  libieee.a       libnss_dns_pic.a    libresolv.a             libutil_p.a
libc_nonshared.a        libm.a          libnss_dns.so       libresolv_p.a           libutil_pic.a
libc_p.a                libmcheck.a     libnss_files_pic.a  libresolv_pic.a         libutil.so
libc_pic.a              libm_p.a        libnss_files.so     libresolv_pic.map
libcrypt.a              libm_pic.a      libnss_hesiod_pic.a libresolv.so
libcrypt_p.a            libm.so         libnss_hesiod.so    librpcsvc.a
zuozhongkai@ubuntu:~/linux/nfs/rootfs/usr/lib$
```

图 38.2.3.4 rootfs/usr/lib 目录

至此，根文件系统的库文件就全部添加好了，可以使用“du”命令来查看一下 rootfs/lib 和 rootfs/usr/lib 这两个目录的大小，命令如下：

```
cd rootfs                   //进入根文件系统目录
du ./lib ./usr/lib/ -sh     //查看 lib 和 usr/lib 这两个目录的大小
```

结果如图 38.2.3.5 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs$ du ./lib ./usr/lib/ -sh
57M     ./lib
67M     ./usr/lib/
zuozhongkai@ubuntu:~/linux/nfs/rootfs$
```

图 38.2.3.5 lib 和 usr/lib 目录大小

可以看出 lib 和 usr/lib 这两个文件的大小分别为 57MB 和 67MB，加起来就是 57+67=124MB。非常大！所以正点原子的 256MB 和 512MB 的 NAND 核心版就不是给初学者准备的，而是给大批量采购的企业准备的，还是那句话，初学者选择 EMMC 版本的。

38.2.4 创建其他文件夹

在根文件系统中创建其他文件夹，如 dev、proc、mnt、sys、tmp 和 root 等，创建完成以后如图 38.2.4.1 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs$ ls
bin  dev  lib  linuxrc  mnt  proc  root  sbin  sys  tmp  usr
zuozhongkai@ubuntu:~/linux/nfs/rootfs$
```

图 38.2.4.1 创建好其他文件夹以后的 rootfs

目前来看，这个根文件系统好像已经准备好了，究竟有没有准备好，直接测一下就知道了！

38.3 根文件系统初步测试

接下来我们使用测试一下前面创建好的根文件系统 rootfs，测试方法就是使用 NFS 挂载，

uboot 里面的 bootargs 环境变量会设置“root”的值，所以我们将 root 的值改为 NFS 挂载即可。在 Linux 内核源码里面有相应的文档讲解如何设置，文档为 Documentation/filesystems/nfs/nfsroot.txt，格式如下：

```
root=/dev/nfs nfsroot=[<server-ip>:]<root-dir>[,<nfs-options>] ip=<client-ip>:<server-ip>:<gw-ip>:<netmask>:<hostname>:<device>:<autoconf>:<dns0-ip>:<dns1-ip>
```

<server-ip>：服务器 IP 地址，也就是存放根文件系统主机的 IP 地址，那就是 Ubuntu 的 IP 地址，比如我的 Ubuntu 主机 IP 地址为 192.168.1.250。

<root-dir>：根文件系统的存放路径，比如我的就是/home/zuozhongkai/linux/nfs/rootfs。

<nfs-options>：NFS 的其他可选选项，一般不设置。

<client-ip>：客户端 IP 地址，也就是我们开发板的 IP 地址，Linux 内核启动以后就会使用此 IP 地址来配置开发板。此地址一定要和 Ubuntu 主机在同一个网段内，并且没有被其他的设备使用，在 Ubuntu 中使用 ping 命令 ping 一下就知道要设置的 IP 地址有没有被使用，如果不能 ping 通就说明没有被使用，那么就可以设置为开发板的 IP 地址，比如我就可以设置为 192.168.1.251。

<server-ip>：服务器 IP 地址，前面已经说了。

<gw-ip>：网关地址，我的就是 192.168.1.1。

<netmask>：子网掩码，我的就是 255.255.255.0。

<hostname>：客户机的名字，一般不设置，此值可以空着。

<device>：设备名，也就是网卡名，一般是 eth0，eth1….，正点原子的 I.MX6U-ALPHA 开发板的 ENET2 为 eth0，ENET1 为 eth1。如果你的电脑只有一个网卡，那么基本只能是 eth0。这里我们使用 ENET2，所以网卡名就是 eth0。

<autoconf>：自动配置，一般不使用，所以设置为 off。

<dns0-ip>：DNS0 服务器 IP 地址，不使用。

<dns1-ip>：DNS1 服务器 IP 地址，不使用。

根据上面的格式 bootargs 环境变量的 root 值如下：

```
root=/dev/nfs rw nfsroot=192.168.1.250:/home/zuozhongkai/linux/nfs/rootfs ip=192.168.1.251:
192.168.1.250:192.168.1.1:255.255.255.0::eth0:off
```

启动开发板，进入 uboot 命令行模式，然后重新设置 bootargs 环境变量，命令如下：

```
setenv bootargs 'console=ttymxc0,115200 root=/dev/nfs rw nfsroot=192.168.1.250:
/home/zuozhongkai/linux/nfs/rootfs ip=192.168.1.251:192.168.1.250:192.168.1.1:255.255.255.0::eth0:
off'                    //设置 bootargs
saveenv                 //保存环境变量
```

设置好以后使用“boot”命令启动 Linux 内核，结果如图 38.3.1 所示：

```
usb 1-1.2: new high-speed USB device number 4 using ci_hdrc
fec 20b4000.ethernet eth0: Link is Up - 100Mbps/Full - flow control rx/tx
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
IP-Config: Complete:
     device=eth0, hwaddr=00:04:9f:04:d2:35, ipaddr=192.168.1.251, mask=255.255.255.0, gw=192.168.1.
     host=192.168.1.251, domain=, nis-domain=(none)
     bootserver=192.168.1.250, rootserver=192.168.1.250, rootpath=
gpio_dvfs: disabling
VSD_3V3: disabling
can-3v3: disabling
ALSA device list:
  #0: wm8960-audio
VFS: Mounted root (nfs filesystem) on device 0:14.
devtmpfs: mounted
Freeing unused kernel memory: 444K (80b19000 - 80b88000)
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console.
/ #
/ #
/ #
/ #
/ #
```

图 38.3.1 进入根文件系统

从图 38.3.1 可以看出，我们进入了根文件系统，说明我们的根文件系统工作了！如果没有启动进入根文件系统的话可以重启一次开发板试试。我们可以输入“ls”命令测试一下，结果如图 38.3.2 所示：

```
/ # ls
bin          lib          mnt          root         sys          usr
dev          linuxrc      proc         sbin         tmp
/ #
```

图 38.3.2 ls 命令测试

可以看出 ls 命令工作正常！那么是不是说明我们的 rootfs 就制作成功了呢？大家注意，在进入根文件系统的时候会有下面这一行错误提示：

```
can't run '/etc/init.d/rcS': No such file or directory
```

提示很简单，说是无法运行“/etc/init.d/rcS”这个文件，因为这个文件不存在。如图 38.3.3 所示

```
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console.
/ #
/ #
```

图 39.3.3 “/etc/init.d/rcS”不存在

看来我们的 rootfs 还是缺文件啊，没什么说的，一步一步的完善吧。

38.4 完善根文件系统

38.4.1 创建/etc/init.d/rcS 文件

rcS 是个 shell 脚本，Linux 内核启动以后需要启动一些服务，而 rcS 就是规定启动哪些文件的脚本文件。在 rootfs 中创建/etc/init.d/rcS 文件，然后在 rcS 中输入如下所示内容：

示例代码 38.4.1.1 /etc/init.d/rcS 文件

```
1  #!/bin/sh
2
3  PATH=/sbin:/bin:/usr/sbin:/usr/bin
4  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib:/usr/lib
5  export PATH LD_LIBRARY_PATH runlevel
```

```
6
7  mount -a
8  mkdir /dev/pts
9  mount -t devpts devpts /dev/pts
10
11 echo /sbin/mdev > /proc/sys/kernel/hotplug
12 mdev -s
```

第 1 行，表示这是一个 shell 脚本。

第 3 行，PATH 环境变量保存着可执行文件可能存在的目录，这样我们在执行一些命令或者可执行文件的时候就不会提示找不到文件这样的错误。

第 4 行，LD_LIBRARY_PATH 环境变量保存着库文件所在的目录。

第 5 行，使用 export 来导出上面这些环境变量，相当于声明一些“全局变量”。

第 7 行，使用 mount 命令来挂载所有的文件系统，这些文件系统由文件/etc/fstab 来指定，所以我们一会还要创建/etc/fstab 文件。

第 8 和 9 行，创建目录/dev/pts，然后将 devpts 挂载到/dev/pts 目录中。

第 11 和 12 行，使用 mdev 来管理热插拔设备，通过这两行，Linux 内核就可以在/dev 目录下自动创建设备节点。关于 mdev 的详细内容可以参考 busybox 中的 docs/mdev.txt 文档。

示例代码 38.4.1.1 中的 rcS 文件内容是最精简的，大家如果去看 Ubuntu 或者其他大型 Linux 操作系统中的 rcS 文件，就会发现其非常复杂。因为我们是初次学习，所以不用搞这么复杂的，而且这么复杂的 rcS 文件也是借助其他工具创建的，比如 buildroot 等。

创建好文件/etc/init.d/rcS 以后一定要给其可执行权限！

创建好文件/etc/init.d/rcS 以后一定要给其可执行权限！

创建好文件/etc/init.d/rcS 以后一定要给其可执行权限！

使用如下命令给予/ec/init.d/rcS 可执行权限：

```
chmod 777 rcS
```

设置好以后就重新启动 Linux 内核，启动以后如图 38.4.1.1 所示：

```
mount: can't read '/etc/fstab': No such file or directory
/etc/init.d/rcS: line 13: can't create /proc/sys/kernel/hotplug: nonexistent directory
mdev: /sys/dev: No such file or directory

Please press Enter to activate this console.
```

图 38.4.1.1 Linux 启动过程

从图 38.4.1.1 可以看到，提示找不到/etc/fstab 文件，还有一些其他的错误，我们先把/etc/fstab 这个错误解决了。说不定把这个问题解决以后其他的错误也就解决了。前面我们说了“mount -a”挂载所有根文件系统的时候需要读取/etc/fstab，因为/etc、fstab 里面定义了改挂载哪些文件，好了，接下来就是创建/etc/fstab 文件。

38.4.2 创建/etc/fstab 文件

在 rootfs 中创建/etc/fstab 文件，fstab 在 Linux 开机以后自动配置哪些需要自动挂载的分区，格式如下：

```
<file system>    <mount point>    <type>    <options>    <dump>    <pass>
```

<file system>：要挂载的特殊的设备，也可以是块设备，比如/dev/sda 等等。

<mount point>：挂载点。

<type>：文件系统类型，比如 ext2、ext3、proc、romfs、tmpfs 等等。

<options>：挂载选项，在 Ubuntu 中输入“man mount”命令可以查看具体的选项。一般使

用 defaults，也就是默认选项，defaults 包含了 rw、suid、 dev、 exec、 auto、 nouser 和 async。

<dump>：为 1 的话表示允许备份，为 0 不备份，一般不备份，因此设置为 0。

<pass>：磁盘检查设置，为 0 表示不检查。根目录‘/’设置为 1，其他的都不能设置为 1，其他的分区从 2 开始。一般不在 fstab 中挂载根目录，因此这里一般设置为 0。

按照上述格式，在 fstab 文件中输入如下内容：

示例代码 38.4.2.1 /etc/fstab 文件

```
1 #<file system>  <mount point>     <type>     <options>    <dump>   <pass>
2 proc            /proc             proc       defaults     0        0
3 tmpfs           /tmp              tmpfs      defaults     0        0
4 sysfs           /sys              sysfs      defaults     0        0
```

fstab 文件创建完成以后重新启动 Linux，结果如图 38.4.2.1 所示：

```
VFS: Mounted root (nfs filesystem) on device 0:14.
devtmpfs: mounted
Freeing unused kernel memory: 444K (80b19000 - 80b88000)

Please press Enter to activate this console.
/ #
```

图 38.4.2.1 Linux 启动过程

从图 38.4.2.1 可以看出，启动启动成功，而且没有任何错误提示。但是我们要需要创建一个文件/etc/inittab。

38.4.3 创建/etc/inittab 文件

inittab 的详细内容可以参考 busybox 下的文件 examples/inittab。init 程序会读取/etc/inittab 这个文件，inittab 由若干条指令组成。每条指令的结构都是一样的，由以“:”分隔的 4 个段组成，格式如下：

```
<id>:<runlevels>:<action>:<process>
```

<id>：每个指令的标识符，不能重复。但是对于 busybox 的 init 来说，<id>有着特殊意义。对于 busybox 而言<id>用来指定启动进程的控制 tty，一般我们将串口或者 LCD 屏幕设置为控制 tty。

<runlevels>：对 busybox 来说此项完全没用，所以空着。

<action>：动作，用于指定<process>可能用到的动作。busybox 支持的动作如表 38.4.3.1 所示：

动作	描述
sysinit	在系统初始化的时候 process 才会执行一次。
respawn	当 process 终止以后马上启动一个新的。
askfirst	和 respawn 类似，在运行 process 之前在控制台上显示“Please press Enter to activate this console.”。只要用户按下“Enter”键以后才会执行 process。
wait	告诉 init，要等待相应的进程执行完以后才能继续执行。
once	仅执行一次，而且不会等待 process 执行完成。
restart	当 init 重启的时候才会执行 procee。
ctrlaltdel	当按下 ctrl+alt+del 组合键才会执行 process。
shutdown	关机的时候执行 process。

表 38.4.3.1 动作

<process>：具体的动作，比如程序、脚本或命令等。

参考 busybox 的 examples/inittab 文件，我们也创建一个/etc/inittab，在里面输入如下内容：

示例代码 38.4.3.1 /etc/inittab 文件

```
1 #etc/inittab
2 ::sysinit:/etc/init.d/rcS
3 console::askfirst:-/bin/sh
4 ::restart:/sbin/init
5 ::ctrlaltdel:/sbin/reboot
6 ::shutdown:/bin/umount -a -r
7 ::shutdown:/sbin/swapoff -a
```

第 2 行，系统启动以后运行/etc/init.d/rcS 这个脚本文件。

第 3 行，将 console 作为控制台终端，也就是 ttymxc0。

第 4 行，重启的话运行/sbin/init。

第 5 行，按下 ctrl+alt+del 组合键的话就运行/sbin/reboot，看来 ctrl+alt+del 组合键用于重启系统。

第 6 行，关机的时候执行/bin/umount，也就是卸载各个文件系统。

第 7 行，关机的时候执行/sbin/swapoff，也就是关闭交换分区。

/etc/initta 文件创建好以后就可以重启开发板即可，至此！根文件系统要创建的文件就已经全部完成了。接下来就要对根文件系统进行其他的测试，比如是我们自己编写的软件运行收费正常、是否支持软件开机自启动、中文支持是否正常以及能不能链接等。

38.5 根文件系统其他功能测试

38.5.1 软件运行测试

我们使用 Linux 的目的就是运行我们自己的软件，我们编译的应用软件一般都使用动态库，使用动态库的话应用软件体积就很小，但是得提供库文件，库文件我们已经添加到了根文件系统中。我们编写一个小小的测试软件来测试一下库文件是否工作正常，在根文件系统下创建一个名为“drivers”的文件夹，以后我们学习 Linux 驱动的时候就把所有的实验文件放到这个文件夹里面。

在 ubuntu 下使用 vim 编辑器新建一个 hello.c 文件，在 hello.c 里面输入如下内容：

示例代码 38.5.1.1 hello.c 文件

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      while(1) {
6          printf("hello world!\r\n");
7          sleep(2);
8      }
9      return 0;
10 }
```

hello.c 内容很简单，就是循环输出“hello world”，sleep 相当于 Linux 的延时函数，单位为秒，所以 sleep(2)就是延时 2 秒。编写好以后就是编译，因为我们是要在 ARM 芯片上运行的，所以要用交叉编译器去编译，也就是使用 arm-linux-gnueabihf-gcc 编译，命令如下：

```
arm-linux-gnueabihf-gcc hello.c -o hello
```

使用 arm-linux-gnueabihf-gcc 将 hello.c 编译为 hello 可执行文件。这个 hello 可执行文件究竟是不是 ARM 使用的呢？使用“file”命令查看文件类型以及编码格式：

```
file hello        //查看 hello 的文件类型以及编码格式
```

结果如图 38.5.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/rootfs/drivers$ arm-linux-gnueabihf-gcc hello.c -o hello
zuozhongkai@ubuntu:~/linux/nfs/rootfs/drivers$ ls
hello  hello.c
zuozhongkai@ubuntu:~/linux/nfs/rootfs/drivers$ file hello
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpr
eter /lib/ld-, for GNU/Linux 2.6.31, BuildID[sha1]=7dd1bde89e09327b11ad95e22e72f9bfafd8aec
b, not stripped
zuozhongkai@ubuntu:~/linux/nfs/rootfs/drivers$
```


图 38.5.1.1 查看 hello 编码格式

从图 38.5.1.1 可以看出，输入“file hello”输入了如下所示信息：

```
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked……
```

hello 是个 32 位的 LSB 可执行文件，ARM 架构的，并且是动态链接的。所以我们编译出来的 hello 文件没有问题。将其拷贝到 rootfs/drivers 目录下，在开发板中输入如下命令来执行这个可执行文件：

```
cd /drivers       //进入 drivers 目录
./hello           //执行 hello
```

结果如图 38.5.1.2 所示：

```
/ # cd drivers/
/drivers # ./hello
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!

/drivers #
```


图 38.5.1.2 hello 运行结果

可以看出，hello 这个软件运行正常，说明我们的根文件系统中的共享库是没问题的，要想终止 hello 的运行，按下“ctrl+c”组合键即可。此时大家应该能感觉到，hello 执行的时候终端是没法用的，除非使用“ctrl+c”来关闭 hello，那么有没有办法既能让 hello 正常运行，而且终端能够正常使用？那肯定是有的，让 hello 进入后台运行就行了，让一个软件进入后台的方法很简单，运行软件的时候加上“&”即可，比如“./hello &”就是让 hello 在后台运行。在后台运行的软件可以使用“kill -9 pid(进程 ID)”命令来关闭掉，首先使用“ps”命令查看要关闭的软件 PID 是多少，ps 命令用于查看所有当前正在运行的进程，并且会给出进程的 PID。输入“ps”命令，结果如图 38.5.1.3 所示：

```
/drivers # ps
PID   USER       TIME   COMMAND
    1 0          0:01 {linuxrc} init
    2 0          0:00 [kthreadd]
    3 0          0:00 [ksoftirqd/0]
    5 0          0:00 [kworker/0:0H]
    6 0          0:00 [kworker/u2:0]
    7 0          0:04 [rcu_preempt]
    8 0          0:00 [rcu_sched]
    9 0          0:00 [rcu_bh]
   10 0          0:00 [migration/0]
   11 0          0:00 [khelper]
   12 0          0:00 [kdevtmpfs]
   13 0          0:00 [perf]
   14 0          0:00 [writeback]
   15 0          0:00 [crypto]
   16 0          0:00 [bioset]
   17 0          0:00 [kblockd]
   18 0          0:00 [ata_sff]
   20 0          0:00 [cfg80211]
   21 0          0:00 [rpciod]
   22 0          0:00 [kswapd0]
   23 0          0:00 [fsnotify_mark]
   24 0          0:00 [nfsiod]
   61 0          0:00 [kworker/u2:1]
   66 0          0:00 [ci_otg]
   67 0          0:00 [irq/21-2040000.]
   68 0          0:00 [cfinteractive]
   69 0          0:00 [irq/225-mmc0]
   71 0          0:00 [irq/50-2190000.]
   72 0          0:00 [irq/226-mmc1]
   73 0          0:00 [mxs_dcp_chan/sh]
   74 0          0:00 [mxs_dcp_chan/ae]
   81 0          0:00 [ipv6_addrconf]
   82 0          0:00 [krfcommd]
   83 0          0:00 [pxp_dispatch]
   84 0          0:00 [deferwq]
   85 0          0:00 [irq/205-imx_the]
   87 0          0:00 [kworker/0:1H]
   93 0          0:00 -/bin/sh
   97 0          0:00 [mmcqd/0]
  103 0          0:00 [mmcqd/1]
  104 0          0:00 [mmcqd/1boot0]
  105 0          0:00 [mmcqd/1boot1]
  106 0          0:00 [mmcqd/1rpmb]
  133 0          0:00 udhcpc -i eth1
  163 0          0:00 [kworker/0:0]
  164 0          0:00 [kworker/0:2]
  165 0          0:00 [kworker/0:1]
  166 0          0:00 ./hello
  168 0          0:00 ps
```

图 38.5.1.3 ps 命令结果

从图 38.5.1.3 可以看出 hello 对应的 PID 为 166，因此我们使用如下命令关闭在后台运行的 hello 软件：

```
kill -9 166
```

因为 hello 在不断的输出“hello world”所以我们的输入看起来会被打断，其实是没有的，因为我们是输入，而 hello 是输出。在数据流上是没有打断的，只是显示在 SecureCRT 上就好像被打断了，所以只管输入“kill -9 166”即可。hello 被 kill 以后会有提示，如图 38.5.1.4 所示：

```
[1]+  Killed                     ./hello
/drivers #
```

图 38.5.1.4 提示 hello 被 kill 掉。

再去用 ps 命令查看一下当前的进程，发现没有 hello 了。这个就是 Linux 下的软件后台运行以及如何关闭软件的方法，重点就是 3 个操作：软件后面加“&”、使用 ps 查看要关闭的软件 PID、使用“kill -9 pid”来关闭指定的软件。

38.5.2 中文字符测试

1、设置 SecureCRT 使用 UTF-8 编码

因为 Linux 使用的编码格式为 UTF-8，因此要先设置 SecureCRT 的编码格式。打开 Options->Session Options…，打开“Session Options”对话框，选择左侧的“Appearance”，然后在右侧的“Character encoding:”栏选择 UTF-8 编码，如图 38.5.2.1 所示：

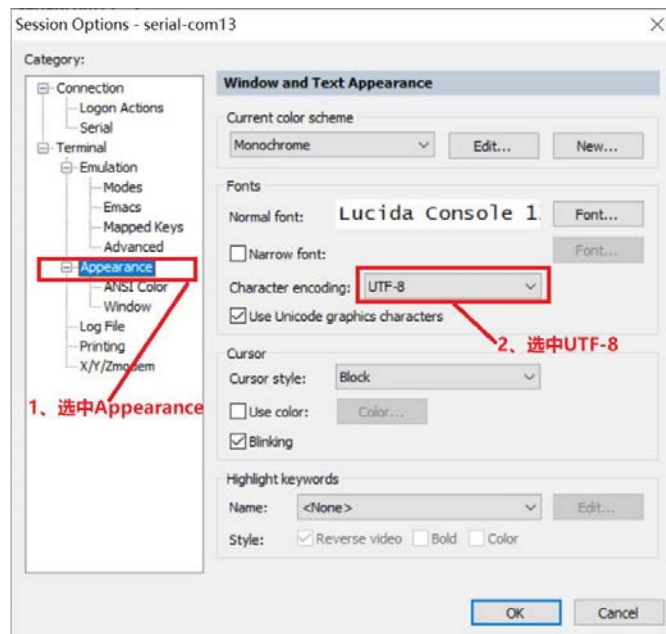


图 38.5.2.1 设置 UTF-8 编码

设置好以后点击下方的“Ok”按钮即可，SecureCRT 我们就设置好了。

2、创建中文文件

在 ubuntu 中向在 rootfs 目录新建一个名为“中文测试”的文件夹，然后在 SecureCRT 下查看中文名能不能显示正确。输入“ls”命令，结果如图 38.5.2.2 所示：

```
/drivers # cd /
/ # ls
bin         etc         mnt         sbin        usr
dev         lib         proc        sys         中文测试
drivers     linuxrc     root        tmp
/ #
```

图 38.5.2.2 中文文件夹测试

可以看出“中文测试”这个文件夹显示正常，接着“touch”命令在“中文测试”文件夹中新建一个名为“测试文档.txt”的文件，并且使用 vim 编辑器在其中输入“这是一个中文测试文件”，借此来测试一下中文文件名和中文内容显示是否正常。在 SecureCRT 中使用“cat”命令来查看“测试文档.txt”中的内容，结果如图 38.5.2.3 所示：

```
/ # cd 中文测试/
/中文测试 # ls
测试文档.txt
/中文测试 # cat 测试文档.txt        1、cat命令查看”测试文档.txt“内容
这是一个中文测试文件                 2、文档内容
/中文测试 #
```

图 38.5.2.3 中文文档内容显示

从图 38.5.2.3 可以看出，“测试文档.txt”的中文内容显示正确，而且中文路径也完全正常，说明我们的根文件系统已经完美支持中文了！

38.5.3 开机自启动测试

在 38.5.1 小节测试 hello 软件的时候都是等 Linux 启动进入根文件系统以后手动输入命令“./hello”来完成的。我们一般做好产品以后都是需要开机自动启动相应的软件，本节我们就以

hello 这个软件为例，讲解一下如何实现开机自启动。前面我们说过了，进入根文件系统的时候会运行/etc/init.d/rcS 这个 shell 脚本，因此我们可以在这个脚本里面添加自启动相关内容。添加完成以后的/etc/init.d/rcS 文件内容如下：

示例代码 38.5.3.1 rcS 文件代码

```
1  #!/bin/sh
2  PATH=/sbin:/bin:/usr/sbin:/usr/bin
3  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib:/usr/lib
4  runlevel=S
5  umask 022
6  export PATH LD_LIBRARY_PATH runlevel
7
8  mount -a
9  mkdir /dev/pts
10 mount -t devpts devpts /dev/pts
11
12 echo /sbin/mdev > /proc/sys/kernel/hotplug
13 mdev -s
14
15 #开机自启动
16 cd /drivers
17 ./hello &
18 cd /
```

第 16 行，进入 drivers 目录，因为要启动的软件存放在 drivers 目录下。

第 17 行，以后台方式执行 hello 这个软件。

第 18 行，退出 drivers 目录，进入到根目录下。

自启动代码添加完成以后就可以重启开发板，看看 hello 这个软件会不会自动运行。结果如图 38.5.3.1 所示：

```
 #0: wm8960-audio
VFS: Mounted root (nfs filesystem) on device 0:14.
devtmpfs: mounted
Freeing unused kernel memory: 444K (80b19000 - 80b88000)
nfs: server 192.168.1.250 not responding, still trying
nfs: server 192.168.1.250 OK

Please press Enter to activate this console. hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
hello world!
```

hello软件开始运行

图 38.5.3.1 hello 开机自启动

从图 38.5.3.1 可以看出，hello 开机自动运行了，说明开机自启动成功。

38.5.4 外网连接测试

这里说的外网不是外国哪些 404 网站的连接测试，而是百度、淘宝等这些网站的测试。也就是说看看我们的开发板能不能上网，能不能和我们的局域网外的这些网站进行通信。测试方法很简单，就是通过 ping 命令来 ping 一下百度的官网：www.baidu.com。输入如下命令：

```
ping www.baidu.com
```

结果如图 38.5.4.1 所示：

```
/ # ping www.baidu.com
ping: bad address 'www.baidu.com'
/ #
```

图 38.5.4.1 ping 测试结果

可以看出，测试失败，提示 www.baidu.com 是个“bad address”，也就是地址不对，显然我们的地址是正确的。之所以出现这个错误提示是因为 www.baidu.com 的地址解析失败了，并没有解析出其对应的 IP 地址。我们需要配置域名解析服务器的 IP 地址，一般域名解析地址可以设置为所处网络的网关地址，比如 192.168.1.1。也可以设置为 114.114.1144.114，这个是运营商的域名解析服务器地址。

在 rootfs 中新建文件/etc/resolv.conf，然后在里面输入如下内容：

示例代码 38.5.4.1 resolv.conf 文件内容

```
1 nameserver 114.114.114.114
2 nameserver 192.168.1.1
```

设置很简单，nameserver 表示这是个域名服务器，设置了两个域名服务器地址：114.114.114.114 和 192.168.1.1，大家也可以改为其他的域名服务器试试。如果使用“udhcpc”命令自动获取 IP 地址，“udhcpc”命令会修改 nameserver 的值，一般是将其设置为对应的网关地址。修改好以后保存退出，重启开发板！重启以后重新 ping 一下百度官网，结果如图 38.5.4.2 所示：

```
/ # ping www.baidu.com
PING www.baidu.com (14.215.177.39): 56 data bytes
64 bytes from 14.215.177.39: seq=0 ttl=56 time=9.591 ms
64 bytes from 14.215.177.39: seq=1 ttl=56 time=10.127 ms
64 bytes from 14.215.177.39: seq=2 ttl=56 time=13.850 ms
64 bytes from 14.215.177.39: seq=3 ttl=56 time=10.878 ms
64 bytes from 14.215.177.39: seq=4 ttl=56 time=9.816 ms
64 bytes from 14.215.177.39: seq=5 ttl=56 time=10.515 ms
^C
--- www.baidu.com ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 9.591/10.796/13.850 ms
/ #
```

图 38.5.4.2 ping 百度官网结果

可以看出 ping 百度官网成功了！域名也成功的解析了，至此！我们的根文件系统就彻底的制作完成，这个根文件系统最好打包保存一下，防止以后做实验不小心破坏了根文件系统而功亏一篑，又得从头制作根文件系统。uboot、Linux kernel、rootfs 这三个共同构成了一个完整的 Linux 系统，现在的系统至少是一个可以正常运行的系统，后面我们就可以在这个系统上完成 Linux 驱动开发的学习。

第三十九章 系统烧写

前面我们已经移植好了 uboot 和 linux kernle，制作好了根文件系统。但是我们移植都是通过网络来测试的，在实际的产品开发中肯定不可能通过网络来运行，否则没网的时候产品岂不是就歇菜了。因此我们需要将 uboot、linux kernel、.dtb(设备树)和 rootfs 这四个文件烧写到板子上的 EMMC、NAND 或 QSPI Flash 等其他存储设备上，这样不管有没有网络我们的产品都可以正常运行。本章我们就来学习一下如何使用 NXP 官方提供的 MfgTool 工具通过 USB OTG 口来烧写系统。

39.1 MfgTool 工具简介

MfgTool 工具是 NXP 提供的专门用于给 I.MX 系列 CPU 烧写系统的软件，可以在 NXP 官网下载到。此工具已经放到了开发板光盘中，路劲为：5、开发工具->3、NXP 官方原版 MFG_TOOL 烧写工具->L4.1.15_2.0.0-ga_mfg-tools.tar.gz。此软件在 Windows 下使用，对于我们来说太友好了。将此压缩包进行解压，解压完成以后会出现一个名为 L4.1.15_2.0.0-ga_mfg-tools 的文件夹，进入此文件夹，此文件夹的内容如图 39.1.1 所示：

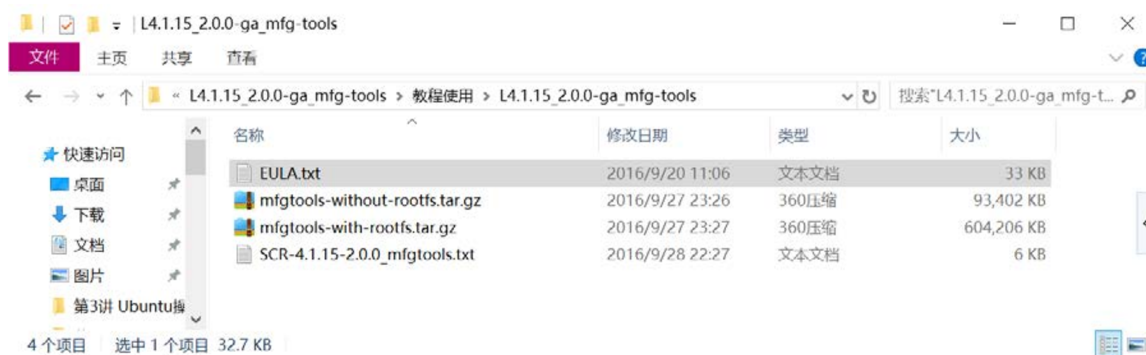


图 39.1.1 mfg_tools 工具目录

从图 39.1.1 可以看出，有两个.txt 文件和两个.gz 压缩包。.txt 文档就不去看了，重点是这两个.gz 压缩包，这两个压缩包的区别在名字上已经写的很详细了。"without-rootfs"和"with-rootfs"，一个是带 rootfs 和一个是不带 rootfs。mfg_tools 这个工具本意是给 NXP 自己的开发板设计的烧写软件，所以肯定带有自家开发板对应的 uboot、linux kernel 和 rootfs 的文件。我们肯定是要烧写文件系统的，所以选择 mfgtools-with-rootfs.tar.gz 这个压缩包，继续对其解压， 解压出一个名为 mfgtools-with-rootfs 的文件夹，此文件夹就包含有我们需要的烧写工具。

进入目录 mfgtools-with-rootfs\mfgtools 中，在此目录下有几个文件夹和很多的.vbs 文件，如图 39.1.2 所示：

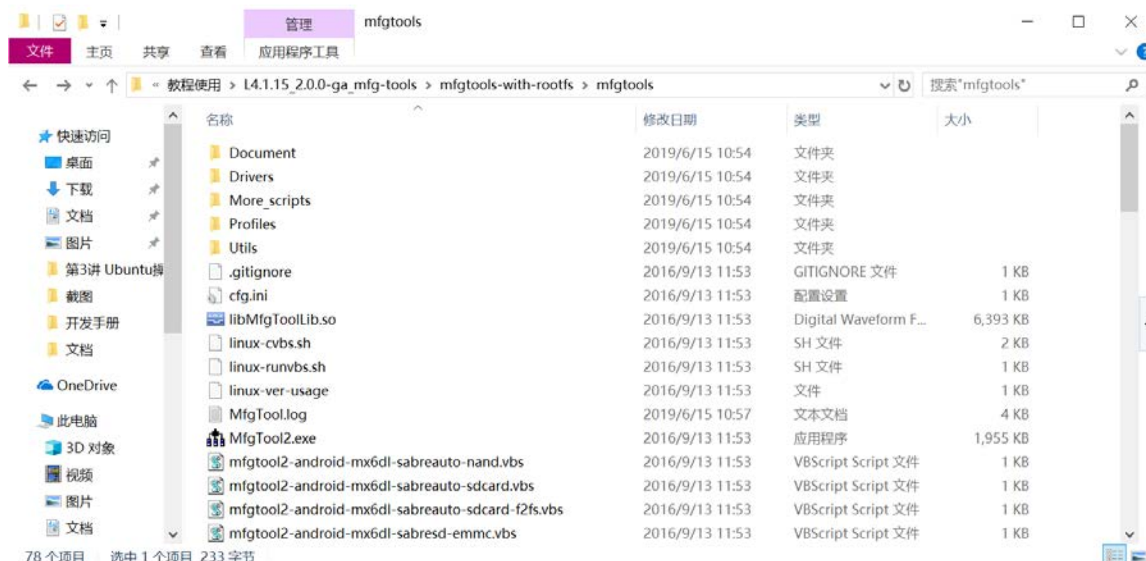


图 39.1.2 mfgtools 目录内容

我们只关心图 39.1.2 中 Profiles 这个文件夹，因为后面要烧写文件就放到这个文件夹中。MfgTool2.exe 就是烧写软件，但是我们不会直接打开这个软件烧写，mfg_tools 不仅能烧写 I.MX6U，而且也能给 I.MX7、I.MX6Q 等芯片烧写，所以在烧写之前必须要进行配置，指定烧

写的是什么芯片，烧写到哪里去？下面的这些众多的.vbs 文件就是配置脚本，烧写的时候通过双击这些.vbs 文件来打开烧写工具。这些.vbs 烧写脚本既可以根据处理器的不同，由用户选择向 I.MX6D、I.MX6Q、I.MX6S、I.MX7、I.MX6UL 和 I.MX6ULL 等的哪一款芯片烧写系统。也可以根据存储芯片的不同，选择向 EMMC、NAND 或 QSPI Flash 等的哪一种存储设备烧写，功能非常强大!！我们现在需要向 I.MX6U 烧写系统，因此需要参考表 39.1.1 所示的 5 个烧写脚本：

脚本文件	描述
mfgtool2-yocto-mx-evk-emmc.vbs	EMMC 烧写脚本。
mfgtool2-yocto-mx-evk-nand.vbs	NAND 烧写脚本
mfgtool2-yocto-mx-evk-qspi-nor-n25q256a.vbs	QSPI Flash 烧写脚本，型号为 n25q256a
mfgtool2-yocto-mx-evk-sdcard-sd1.vbs	如果 SD1 和 SD2 接的 SD 卡，这两个文件分别向 SD1 和 SD2 上的 SD 卡烧写系统。
mfgtool2-yocto-mx-evk-sdcard-sd2.vbs	

表 39.1.1 I.MX6U 使用的烧写脚本

其他的.vbs 烧写脚本用不到，因此可以删除掉，防止干扰我们的视线。本书用的是正点原子的 EMMC 版核心板，因此只会用到 mfgtool2-yocto-mx-evk-emmc.vbs 这个烧写脚本，如果用其他的核心板请参考相应的烧写脚本。

39.2 MfgTool 工作原理简介

MfgTool 只是个工具，具体的原理不需要去深入研究，大概来了解一下其工作原理就行了，知道它的工作流程就行了。

39.2.1 烧写方式

1、连接 USB 线

MfgTool 是通过 USB OTG 接口将系统烧写进 EMMC 中的，正点原子 I.MX6U-ALPHA 开发板上的 USB OTG 口如图 39.2.1.1 所示：

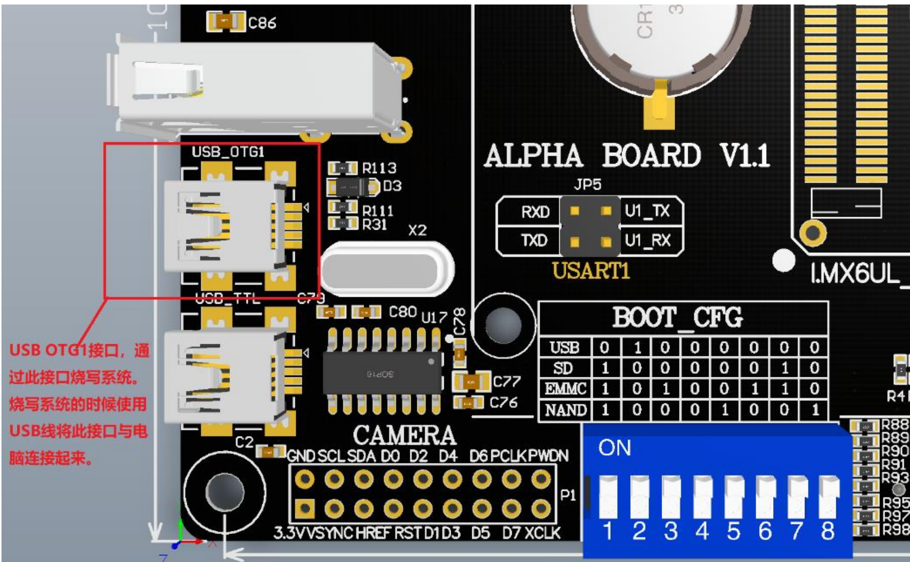


图 39.2.2.1 USB OTG1 接口

在烧写之前，需要先用 USB 线将图 39.2.2.1 中的 USB_OTG1 接口与电脑连接起来。

2、拨码开关拨到 USB 下载模式

将图 39.2.2.1 中的拨码开关拨到“USB”模式，如图 39.2.2.2 所示：

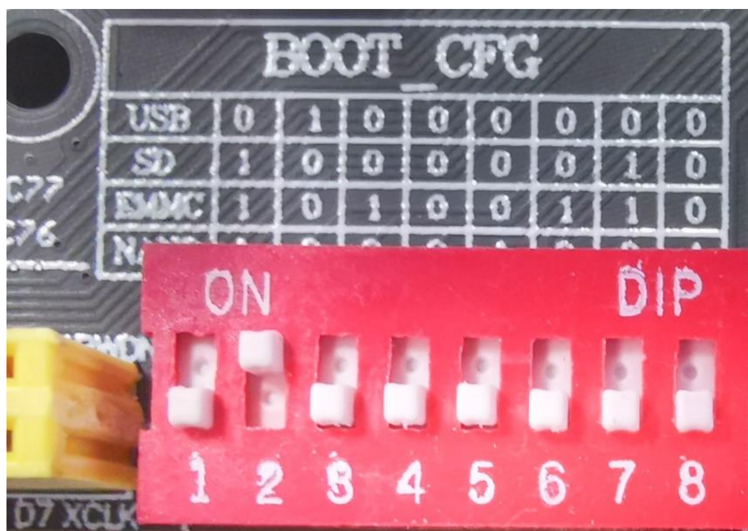


图 39.2.2.2 USB 下载模式

如果插了 TF 卡，请弹出 TF 卡，否则电脑不能识别 USB！等识别出来以后再插上 TF 卡！
如果插了 TF 卡，请弹出 TF 卡，否则电脑不能识别 USB！等识别出来以后再插上 TF 卡！
如果插了 TF 卡，请弹出 TF 卡，否则电脑不能识别 USB！等识别出来以后再插上 TF 卡！

一切准备就绪以后，按一下开发板的复位键，此时就会进入到 USB 模式，如果是第一次进入 USB 模式的话可能会久一点，这个是免驱的，因此不需要安装驱动。第一次进入 USB 模式会在电脑右下角有如图 39.2.2.3 所示提示：

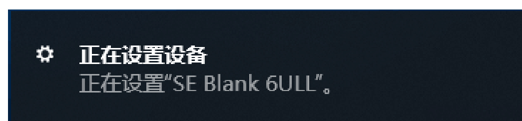


图 39.2.2.3 第一次进入 USB 模式

一旦第一次设置好设备以后，后面每次连接都不会有任何提示了。到这里，我们的开发板已经和电脑连接好了，可以开始烧写系统了。

39.2.2 系统烧写原理

开发板连接电脑以后双击“mfgtool2-yocto-mx-evk-emmc.vbs”，打开下载对话框，如图 39.2.2.1 所示：



图 39.2.2.1 MfgTool 工具界面

如果出现“符合 HID 标准的供应商定义设备”就说明连接正常，可以进行烧写，如果出现其他的字符那么就要检查连接是否正确。点击“Start”按钮即可开始烧写，烧写什么东西呢？肯定是烧写 uboot、Linux kernel、.dtb 和 rootfs，那么这四个应该放到哪里 MfgTool 才能访问到呢？进入如下目录中：

```
L4.1.15_2.0.0-ga_mfg-tools/mfgtools-with-rootfs/mfgtools/Profiles/Linux/OS Firmware
```

此目录中的文件如图 39.2.2.2 所示：

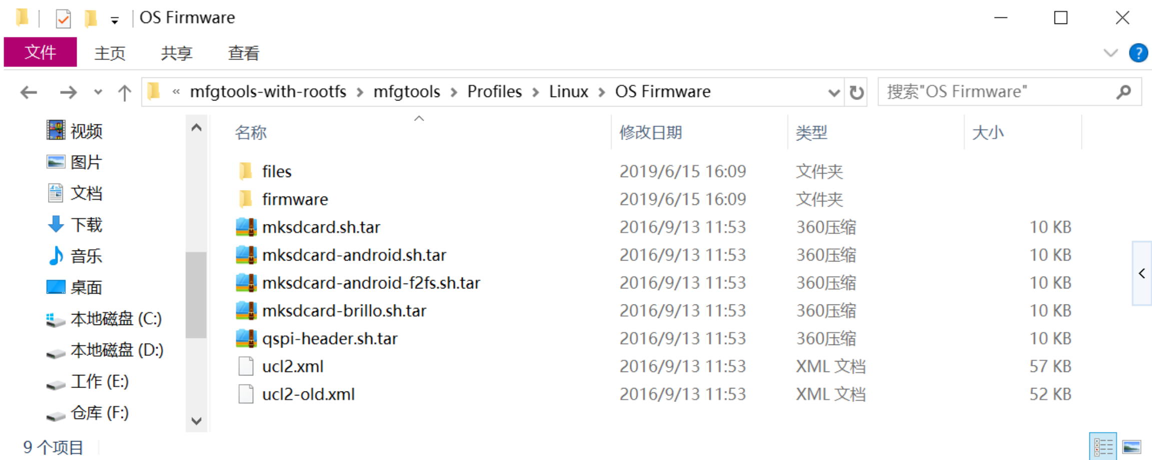


图 39.2.2.2 OS Firmware 文件夹内容

文件夹“OS Firmware”看名字就知道是存放系统固件的，我们重点关注 files、firmware 这两个文件夹，以及 ucl2.xml 这个文件。在具体看这三个文件和文件夹之前，我们先来简单了解一下 MfgTool 烧写的原理，MfgTool 其实是先通过 USB OTG 先将 uboot、kernel 和.dtb(设备树)这是三个文件下载到开发板的 DDR 中，注意不需要下载 rootfs。就相当于直接在开发板的 DDR 上启动 Linux 系统，等 Linux 系统启动以后再向 EMMC 中烧写完整的系统，包括 uboot、linux kernel、.dtb(设备树)和 rootfs，因此 MfgTool 工作过程主要分两个阶段：

①、将 firmware 目录中的 uboot、linux kernel 和.dtb(设备树)，然后通过 USB OTG 将这个文件下载到开发板的 DDR 中，目的就是在 DDR 中启动 Linux 系统，为后面的烧写做准备。

②、经过第①步的操作，此时 Linux 系统已经运行起来了，系统运行起来以后就可以很方便的完成对 EMMC 的格式化、分区等操作。EMMC 分区建立好以后就可以从 firmware 中读取要烧写的 uboot、linux kernel、.dtb(设备树)和 rootfs 这 4 个文件，然后将其烧写到 EMMC 中，这个就是 MfgTool 的大概工作流程。

1、firmeare 文件夹

打开 firmware 文件夹，里面有很多的.imx 结尾的 uboot 文件、一个 zImage 镜像文件、很多.dtb 结尾的设备树文件。这些文件都是 NXP 官方开发板使用的，不同的板子使用不同的文件，其中我们需要关心的只有表 39.2.2.1 中的这三个文件：

脚本文件	描述
zImage	NXP 官方 I.MX6ULL EVK 开发板的 Linux 镜像文件。
u-boot-imx6ull14x14evk_emmc.imx	NXP 官方 I.MX6ULL EVK 开发板的 uboot 文件。
zImage-imx6ull-14x14-evk-emmc.dtb	NXP 官方 I.MX6ULL EVK 开发板的设备树

表 39.2.2.1 I.MX6ULL EVK 开发板使用的系统文件

表 39.2.2.1 中的这三个文件就是 I.MX6ULL EVK 开发板烧写系统的时候第一阶段所需的文件。如果要烧写我们的系统，就需要用我们编译出来的 zImage、u-boot.imx 和 imx6ull-alientek-emmc.dtb 这三个文件替换掉表 39.2.2.1 中这三个文件。但是名字要和表 39.2.2.1 中的一致，因此需要将 u-boot.imx 重命名为 u-boot-imx6ull14x14evk_emmc.imx，将 imx6ull-alientek-emmc.dtb 重命名为 zImage-imx6ull-14x14-evk-emmc.dtb。

2、files 文件夹

将表 39.2.2.1 中的这三个文件下载到开发板的 DDR 上以后烧写的第一阶段就完成了，第二

阶段就是从 files 目录中读取整个系统文件，并将其烧写到 EMMC 中。files 目录中的文件和 firmware 目录中的基本差不多，都是不同板子对应的 uboot、设备树文件，同样，我们只关心表 39.2.2.2 中的四个文件：

脚本文件	描述
zImage	NXP 官方 I.MX6ULL EVK 开发板的 Linux 镜像文件。
u-boot-imx6ull14x14evk_emmc.imx	NXP 官方 I.MX6ULL EVK 开发板的 uboot 文件。
zImage-imx6ull-14x14-evk-emmc.dtb	NXP 官方 I.MX6ULL EVK 开发板的设备树
rootfs_nogpu.tar.bz2	根文件系统，注意和另外一个 rootfs.tar.bz2 根文件系统区分开。nogpu 表示此根文件系统不包含 GPU 的内容，I.MX6ULL 没有 GPU，因此要使用此根文件系统

表 39.2.2.2 I.MX6ULL EVK 开发板烧写文件

如果要烧写我们自己编译出来的系统，就需要用我们编译出来的 zImage、u-boot.imx 和 imx6ull-alientek-emmc.dtb 和 rootfs 这四个文件替换掉表 39.2.2.2 中这四个文件。

3、ucl2.xml 文件

files 和 firmware 目录下有众多的 uboot 和设备树，那么烧写的时候究竟选择哪一个呢？这个工作就是由 ucl2.xml 文件来完成的。ucl2.xml 以“<UCL>”开始，以“</UCL>”结束。“<CFG>”和“</CFG>”之间是配置相关内容，主要是判断当前是给 I.MX 系列的哪个芯片烧写系统。“<LIST>”和“</LIST>”之间的是针对不同存储芯片的烧写命令。整体框架如下：

示例代码 39.2.2.1 ucl2.xml 框架

```
<UCL>
    <CFG>
    ......
    <!-- 判断向 I.MX 系列的哪个芯片烧写系统 -->
    ......
    </CFG>

    <LIST name="SDCard" desc="Choose SD Card as media">
    <!-- 向 SD 卡烧写 Linux 系统 -->
    </LIST>

    <LIST name="eMMC" desc="Choose eMMC as media">
    <!-- 向 EMMC 烧写 Linux 系统 -->
    </LIST>

    <LIST name="Nor Flash" desc="Choose Nor flash as media">
    <!-- 向 Nor Flash 烧写 Linux 系统 -->
    </LIST>

    <LIST name="Quad Nor Flash" desc="Choose Quad Nor flash as media">
    <!-- 向 Quad Nor Flash 烧写 Linux 系统 -->
    </LIST>
```

```
    <LIST name="NAND Flash" desc="Choose NAND as media">
    <!-- 向 NAND Flash 烧写 Linux 系统 -->
    </LIST>

    <LIST name="SDCard-Android" desc="Choose SD Card as media">
    <!-- 向 SD 卡烧写 Android 系统 -->
    </LIST>

    <LIST name="eMMC-Android" desc="Choose eMMC as media">
    <!-- 向 EMMC 烧写 Android 系统 -->
    </LIST>

    <LIST name="Nand-Android" desc="Choose NAND as media">
    <!-- 向 NAND Flash 烧写 Android 系统 -->
    </LIST>

    <LIST name="SDCard-Brillo" desc="Choose SD Card as media">
    <!-- 向 SD 卡烧写 Brillo 系统 -->
    </LIST>
</UCL>
```

ucl2.xml 首先会判断当前要向 I.MX 系列的哪个芯片烧写系统，代码如下：

示例代码 39.2.2.2 判断要烧写的处理器型号

```
21 <CFG>
22   <STATE name="BootStrap" dev="MX6SL" vid="15A2" pid="0063"/>
23   <STATE name="BootStrap" dev="MX6D" vid="15A2" pid="0061"/>
24   <STATE name="BootStrap" dev="MX6Q" vid="15A2" pid="0054"/>
25   <STATE name="BootStrap" dev="MX6SX" vid="15A2" pid="0071"/>
26   <STATE name="BootStrap" dev="MX6UL" vid="15A2" pid="007D"/>
27   <STATE name="BootStrap" dev="MX7D" vid="15A2" pid="0076"/>
28   <STATE name="BootStrap" dev="MX6ULL" vid="15A2" pid="0080"/>
29   <STATE name="Updater"   dev="MSC" vid="066F" pid="37FF"/>
30 </CFG>
```

通过读取芯片的 VID 和 PID 即可判断出当前要烧写什么处理器的系统，如果 VID=0X15A2，PID=0080，那么就表示要给 I.MX6ULL 烧写系统。确定了处理器以后就要确定向什么存储设备烧写系统，这个时候就要有请 mfgtool2-yocto-mx-evk-emmc.vbs 再次登场，此文件内容如下：

示例代码 39.2.2.3 mfgtool2-yocto-mx-evk-emmc.vbs 文件内容

```
Set wshShell = CreateObject("WScript.shell")
wshShell.run "mfgtool2.exe -c ""linux"" -l ""eMMC"" -s
""board=sabresd"" -s ""mmc=1"" -s ""6uluboot=14x14evk"" -s
""6uldtb=14x14-evk"""
Set wshShell = Nothing
```

重点是“wshShell.run”这一行，这里一行调用了 mfgtool2.exe 这个软件，并且还给出了一堆的参数，其中就有“eMMC”字样，说明是向 EMMC 烧写系统，要烧写的存储设备就这样确

定下来了。“wshShell.run”后面还有一堆的其他参数，这些参数都有对应的值，如下所示：

```
board=sabresd
mmc=1
6uluboot=14x14evk
6uldtb=14x14-evk
```

我们继续回到 ucl2.xml 中，既然现在已经知道了是向 I.MX6ULL 的 EMMC 中烧写系统，那么直接在 ucl2.xml 中找到相应的烧写命令就行了，因为相应的命令太长，为了缩小篇幅，我们就以 uboot 的烧写为例讲解一下。前面说了烧写分两个阶段，第一步是通过 USB OTG 向 DDR 中下载系统，第二步才是正常的烧写。通过 USB OTG 向 DDR 下载 uboot 的命令如下：

示例代码 39.2.2.4 通过 USB OTG 下载 uboot

```
<CMD state="BootStrap" type="boot" body="BootStrap" file ="firmware/u-boot-imx6ul%lite%%6uluboot%_emmc.imx" ifdev="MX6ULL">Loading U-boot
</CMD>
```

上面的命令就是 BootStrap 阶段，也就是第一阶段，“file”表示要下载的文件位置，在 firmware 目录下，文件名字为

```
u-boot-imx6ul%lite%%6uluboot%_emmc.imx
```

在 L4.1.15_2.0.0-ga_mfg-tools\mfgtools-with-rootfs\mfgtools-with-rootfs\mfgtools 下找到 cfg.ini 文件，该文件里包含了开发板的一些信息，查看 cfg.ini 文件可得 lite=l 以及一些字符串代表的值。

“%lite%”和“%6uluboot%”分别表示取 lite 和 6uluboot 的值，而 lite=l，6uluboot=14x14evk，因此将这来个值带进去以后就是：

```
u-boot-imx6ull14x14evk _emmc.imx
```

所以，这里向 DDR 中下载的是 firmware/ u-boot-imx6ull14x14evk _emmc.imx 这个 uboot 文件。同样的方法将.dtb(设备树)和 zImage 都下载到 DDR 中以后就会跳转去运行 OS，这个时候会在 MfgTool 工具中会有“Jumping to OS image”提示语句，ucl2.xml 中的跳转命令如下：

示例代码 39.2.2.5 跳转到 OS

```
<CMD state="BootStrap" type="jump" > Jumping to OS image. </CMD>
```

启动 Linux 系统以后就可以在 EMMC 上创建分区，然后烧写 uboot、zImage、.dtb(设备树)和根文件系统。

这个就是 MfgTool 的整个烧写原理，弄懂了烧写原理以后就可以开始试着先将 NXP 官方的系统烧写到正点原子的 I.MX6U-ALPHA 开发板中。

39.3 烧写 NXP 官方系统

我们先试着将 NXP 官方的系统烧写到正点原子的 I.MX6U-ALPHA 开发板中，主要是先熟悉一下烧写过程。因为正点原子的 EMMC 核心版用的也是 512MB 的 DDR3 加 4G 的 EMMC，因此烧写 NXP 官方的系统是没有任何问题的。烧写步骤如下：

①、连接好 USB，拨码开关拨到 USB 下载模式。

②、弹出 TF 卡，然后按下开发板复位按键。

③、打开 SecureCRT。

③、双击“mfgtool2-yocto-mx-evk-emmc.vbs”，打开下载软件，如果出现“符合 HID 标准的供应商定义设备”等字样就说明下载软件已经准备就绪。点击“Start”按钮开发烧写 NXP 官方系统，烧写过程如图 39.3.1 所示：

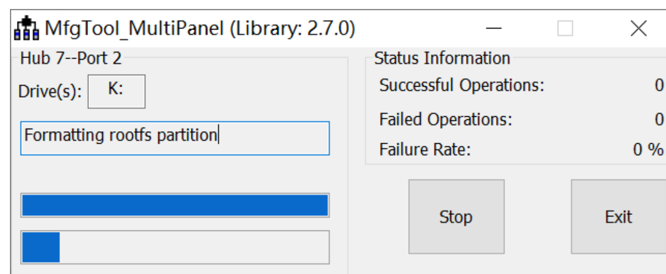


图 39.3.1 烧写过程

这个时候可以在 SecurCRT 上看到具体的烧写过程，如图 39.3.2 所示：

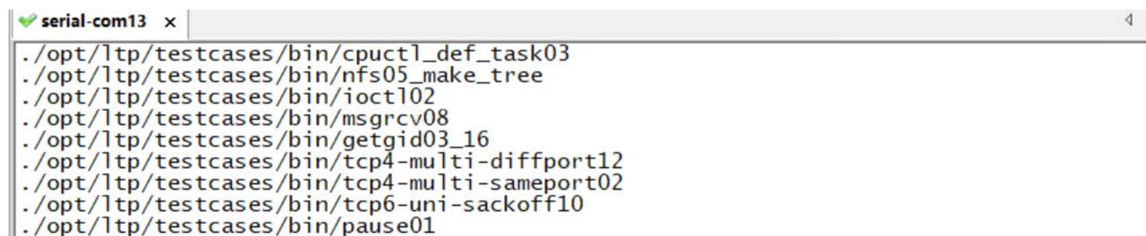


图 39.3.2 正在烧写的文件

等待烧写完成，因为 NXP 官方的根文件系统比较大，因此烧写的时候耗时会久一点。烧写完成以后 MfgTool 软件如图 39.3.3 所示：

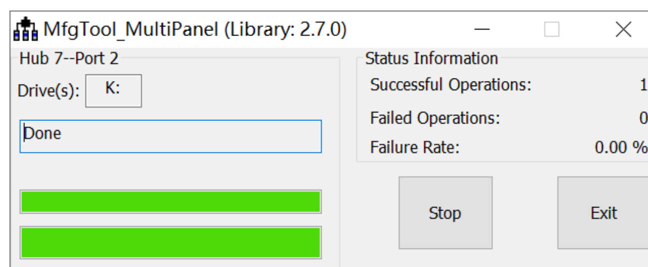


图 39.3.2 烧写完成

烧写完成以后点击“Stop”按钮停止烧写，然后点击“Exit”键退出。拔出 USB 线，将开发板上的拨码开关拨到 EMMC 启动模式，然后重启开发板，此时就会从 EMMC 启动。只是启动以后的系统是 NXP 官方给 I.MX6ULL EVK 开发板制作的，这个系统需要输入用户名，用户名为“root”，没有密码，如图 39.3.3 所示：

```
Running local boot scripts (/etc/rc.local).

Freescale i.MX Release Distro 4.1.15-2.0.0 imx6ul7d /dev/ttymxc0

imx6ul7d login:
```

图 39.3.3 NXP 官方根文件系统

在“imx6ul7d login：”后面输入“root”用户名，然后点击回车键即可进入系统中，进入系统以后就可以进行其他操作了。所以说，NXP 官方的系统其实是可以在正点原子的 EMMC 版核心板上运行的。

39.4 烧写自制的系统

39.4.1 系统烧写

上一小节我们试着将 NXP 官方提供的系统烧写到正点原子的 I.MX6U-ALPHA 开发板好中，目的是体验一下通过 MfgTool 烧写系统的过程。本小节我们就来学习如何将我们做好的系

统烧写到开发板中，首先是准备好要烧写的原材料：

①、自己移植编译出来的 uboot 可执行文件：u-boot.imx。

②、自己移植编译出来的 zImage 镜像文件和开发板对应的.dtb(设备树)，对于 I.MX6U-ALPHA 开发板来说就是 imx6ull-alientek-emmc.dtb。

③、自己构建的根文件系统 rootfs，这里我们需要对 rootfs 进行打包，进入到 Ubuntu 中的 rootfs 目录中，然后使用 tar 命令对其进行打包，命令如下：

```
cd rootfs/
tar -vcjf rootfs.tar.bz2 *
```

完成以后会在 rootfs 目录下生成一个名为 rootfs.tar.bz2 的压缩包，将 rootfs.tar.bz2 发送到 windows 系统中。

将上面提到的这 4 个“原材料”都发送到 Windows 系统中，如图 39.4.1 所示：

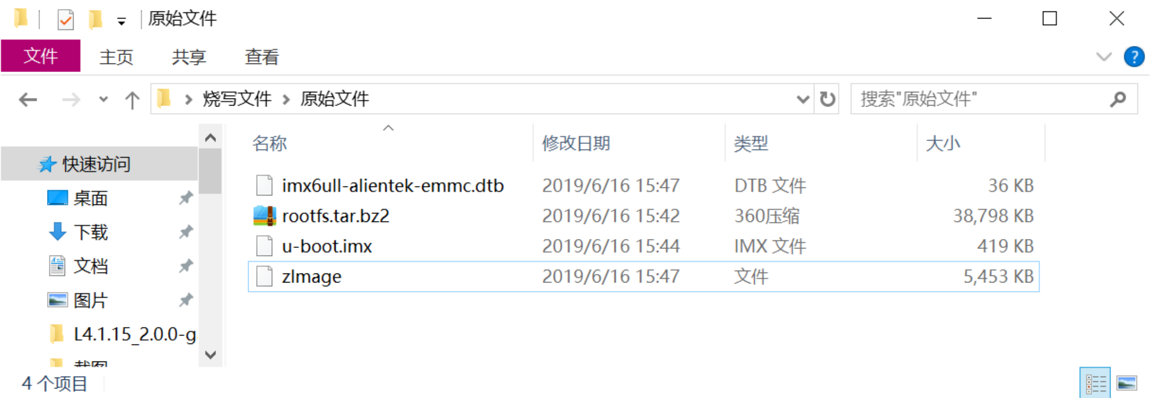


图 39.4.1 烧写原材料

材料准备好以后还不能直接进行烧写，必须对其进行重命名，否则的话 ucl2.xml 是识别不出来的，前面讲解 ucl2.xml 语法的时候已经说过了，图 39.4.1 中的这四个文件重命名见表 39.4.1：

原名字	重命名
u-boot.imx	u-boot-imx6ull14x14evk_emmc.imx
zImage	zImage(不需要重命名)
imx6ull-alientek-emmc.dtb	zImage-imx6ull-14x14-evk-emmc.dtb
rootfs.tar.bz2	rootfs_nogpu.tar.bz2

表 39.4.1 文件重命名表

完成以后如图 39.4.2 所示：

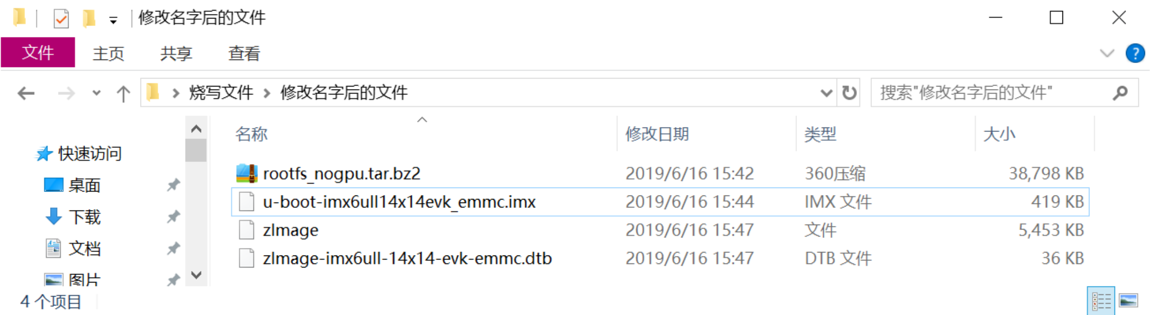


图 39.4.2 重命名以后的文件

接下来就是用我们的文件替换掉 NXP 官方的文件，先将图 39.4.2 中的 zImage、u-boot-imx6ull14x14evk_emmc.imx 和 zImage-imx6ull-14x14-evk-emmc.dtb 这三个文件拷贝到 mfgtools-with-rootfs/mfgtools/Profiles/Linux/OS Firmware/firmware 目录中，替换掉原来的文件。然后将图

39.4.2 中的所有 4 个文件都拷贝到 mfgtools-with-rootfs/mfgtools/Profiles/Linux/OS Firmware/files 目录中，这两个操作完成以后我们就可以进行烧写了。

双击“mfgtool2-yocto-mx-evk-emmc.vbs”，打开烧写软件，点击“Start”按钮开始烧写，由于我们自己制作的 rootfs 比较小，因此烧写相对来说会快一点。烧写完成以后设置开发板从 EMMC 启动，启动我们刚刚烧写进去的系统，测试有没有问题，一般肯定没问题，因为这些都是我们已经测试好的。

39.4.2 网络开机自启动设置

大家在测试网络的时候可能会发现网络不能用，这并不是因为我们将系统烧写到 EMMC 中以后网络坏了。仅仅是因为网络没有打开，我们用 NFS 挂载根文件系统的时候因为要使用 NFS 服务，因此 Linux 内核会打开 eth0 这个网卡，现在我们不使用 NFS 挂载根文件系统，因此 Linux 内核也就不会自动打开 eth0 网卡了。我们可以手动打开网卡，首先输入“ifconfig -a”命令查看一下 eth0 和 eth1 是否都存在，结果如图 39.4.3 所示：

```
/ # ifconfig -a
eth0      Link encap:Ethernet  HWaddr 46:F9:14:3B:59:62
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 2A:61:50:01:A4:3C
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          LOOPBACK  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

图 39.4.3 查看网络

可以看出 eth0 好 eth1 都存在，既然存在我们就打开，以打开 eth0 网卡为例，输入如下命令打开 eth0:

```
ifconfig eth0 up
```

打开网卡的时候会有如图 39.4.4 所示的提示信息：

```
/ # ifconfig eth0 up
fec 20b4000.ethernet eth0: Freescale FEC PHY driver [SMSC LAN8710/LAN8720
] (mii_bus:phy_addr=20b4000.ethernet:01, irq=-1)
IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
/ # fec 20b4000.ethernet eth0: Link is Up - 100Mbps/Full - flow control r
x/tx
IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
```

图 39.4.5 打开 eth0 网卡

打开的时候会提示使用 LAN8710/LAN8720 的网络芯片，eth0 连接成功，并且是 100Mpbs 全双工，eth0 链接准备就绪。这个时候输入“ifconfig”命令就会看到 eth0 这个网卡，如图 39.4.6

所示：

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 46:F9:14:3B:59:62
          inet6 addr: fe80::44f9:14ff:fe3b:5962/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:588 errors:0 dropped:24 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:44330 (43.2 KiB)  TX bytes:680 (680.0 B)

/ #
```

图 39.4.6 当前工作的网卡

接下来就是个 eth0 设置 IP 地址，如果你的开发板连接的路由器，那么可以通过路由器自动分配 IP 地址，命令如下：

```
udhcpc -i eth0        //通过路由器分配 IP 地址
```

如果你的开发板连接着电脑，那么就可以手动设置 IP 地址，比如设置为 192.168.1.251，命令如下：

```
ifconfig eth0 192.168.1.251  netmask 255.255.255.0        //设置 IP 地址和子网掩码
route add default gw 192.168.1.1                          //添加默认网关
```

推荐大家将开发板连接到路由器上，设置好 IP 地址以后就可以测试网络了，比如 ping 一下电脑 IP 地址，或者 ping 一下百度官网。

每次开机以后都要自己手动打开网卡，然后手动设置 IP 地址也太麻烦了，有没有开机以后自动启动网卡并且设置 IP 地址的方法呢？肯定有的，我们将打开网卡，设置网卡 IP 地址的命令添加到/etc/init.d/rcS 文件中就行了，完成以后的 rcS 文件内容如下所示：

```
//示例代码 39.4.2.1 网络开机自启动
1  #!/bin/sh
2
3  PATH=/sbin:/bin:/usr/sbin:/usr/bin
4  LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/lib:/usr/lib
5  export PATH LD_LIBRARY_PATH runlevel
6
7  #网络开机自启动设置
8  ifconfig eth0 up
9  #udhcpc -i eth0
10 ifconfig eth0 192.168.1.251 netmask 255.255.255.0
11 route add default gw 192.168.1.1
......
12 #cd /drivers
13 #./hello &
14 #cd /
```

第 8 行，打开 eth0 网卡

第 9 行，通过路由器自动获取 IP 地址。

第 10 行，手动设置 eth0 的 IP 地址和子网掩码。

第 11 行，添加默认网关。

修改好 rcS 文件以后保存并退出，重启开发板，这个时候 eth0 网卡就会在开机的时候自动启动了，我们也就不用手动添加相关设置了。

39.5 改造我们自己的烧写工具

39.5.1 改造 MfgTool

在上一小节中我们已经实现了将自己的系统烧写到开发板中，但是使用的是“借鸡生蛋”的方法。我们通过将 NXP 官方的系统更换成我们自己制作的系统来完成系统烧写，本节我们就来学习一下如何将 MfgTool 这个工具改造成我们自己的工具，让其支持我们自己的开发板。要改造 MfgTool，重点是三方面：

①、针对不同的核心版，确定系统文件相关名字。

②、新建我们自己的.vbs 文件。

③、修改 ucl2.xml 文件。

1、确定系统文件名字

确定系统文件名字完全是为了兼容不同的产品，比如某个产品有 NAND 和 EMMC 两个版本，那么 EMMC 和 NAND 这两个版本的 uboot、zImage、.dtb 和 rootfs 有可能不同。为了在 MfgTool 工具中同时支持 EMMC 和 NAND 这两个版本的核心板，EMMC 版本的系统文件命名如图 39.5.1.1 所示：

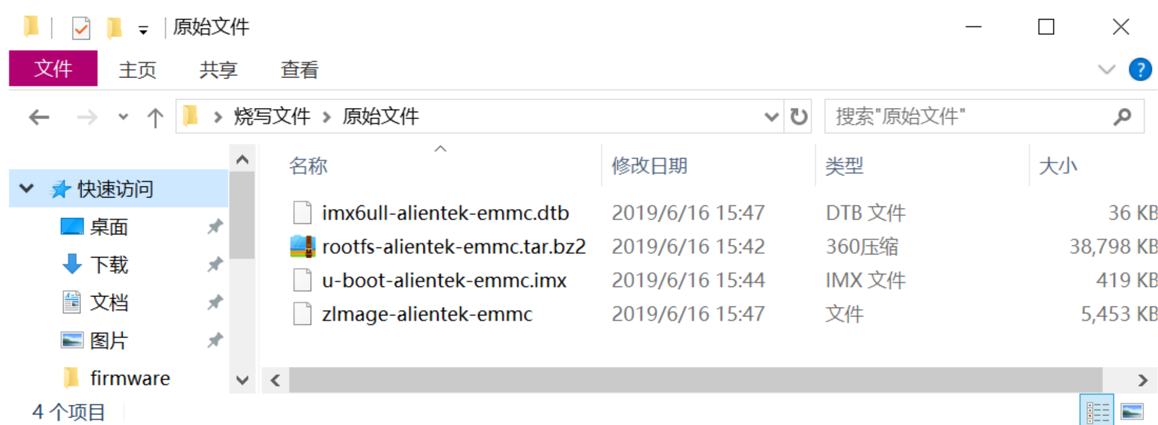


图 39.5.1.1 系统文件名

2、新建.vbs 文件

直接复制 mfgtool2-yocto-mx-evk-emmc.vbs 文件即可，将新复制的文件重命名为 mfgtool2-alientek-alpha-emmc.vbs，文件内容不要做任何修改，.vbs 文件我们就新建好了。

3、修改 ucl2.xml 文件

在修改 ucl2.xml 文件之前，先保存一份原始的 ucl2.xml。将 ucl2.xml 文件改为如下所示内容：

```
<!-- 正点原子修改后的 ucl2.xml 文件 -->

<UCL>
  <CFG>
    <STATE name="BootStrap" dev="MX6UL" vid="15A2" pid="007D"/>
    <STATE name="BootStrap" dev="MX6ULL" vid="15A2" pid="0080"/>
    <STATE name="Updater"   dev="MSC" vid="066F" pid="37FF"/>
  </CFG>

```

```
  <!-- 向 EMMC 烧写系统 -->
  <LIST name="eMMC" desc="Choose eMMC as media">
    <CMD state="BootStrap" type="boot" body="BootStrap" file
="firmware/u-boot-alientek-emmc.imx" ifdev="MX6ULL">Loading U-
boot</CMD>
    <CMD state="BootStrap" type="load" file="firmware/zImage-alientek-
emmc" address="0x80800000"
        loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE"
ifdev="MX6SL MX6SX MX7D MX6UL MX6ULL">Loading Kernel.</CMD>
    <CMD state="BootStrap" type="load" file="firmware/%initramfs%"
address="0x83800000"
        loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE"
ifdev="MX6SL MX6SX MX7D MX6UL MX6ULL">Loading Initramfs.</CMD>
    <CMD state="BootStrap" type="load" file="firmware/imx6ull-alientek-
emmc.dtb" address="0x83000000"
        loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE"
ifdev="MX6ULL">Loading device tree.</CMD>
    <CMD state="BootStrap" type="jump" > Jumping to OS image. </CMD>

    <!-- create partition -->
    <CMD state="Updater" type="push" body="send"
file="mksdcard.sh.tar">Sending partition shell</CMD>
    <CMD state="Updater" type="push" body="$ tar xf $FILE ">
Partitioning...</CMD>
    <CMD state="Updater" type="push" body="$ sh mksdcard.sh
/dev/mmcblk%mmc%"> Partitioning...</CMD>

    <!-- burn uboot -->
    <CMD state="Updater" type="push" body="$ dd if=/dev/zero
of=/dev/mmcblk%mmc% bs=1k seek=768 conv=fsync count=8">clear u-boot
arg</CMD>
    <!-- access boot partition -->
    <CMD state="Updater" type="push" body="$ echo 0 >
/sys/block/mmcblk%mmc%boot0/force_ro">access boot partition 1</CMD>
    <CMD state="Updater" type="push" body="send" file="files/u-boot-
alientek-emmc.imx" ifdev="MX6ULL">Sending u-boot.bin</CMD>
    <CMD state="Updater" type="push" body="$ dd if=$FILE
of=/dev/mmcblk%mmc%boot0 bs=512 seek=2">write U-Boot to sd card</CMD>
    <CMD state="Updater" type="push" body="$ echo 1 >
/sys/block/mmcblk%mmc%boot0/force_ro"> re-enable read-only access
</CMD>
    <CMD state="Updater" type="push" body="$ mmc bootpart enable 1 1
/dev/mmcblk%mmc%">enable boot partion 1 to boot</CMD>
```

```
    <!-- create fat partition -->
    <CMD state="Updater" type="push" body="$ while [ ! -e
/dev/mmcblk%mmc%p1 ]; do sleep 1; echo \"waiting...\"; done ">Waiting
for the partition ready</CMD>
    <CMD state="Updater" type="push" body="$ mkfs.vfat
/dev/mmcblk%mmc%p1">Formatting rootfs partition</CMD>
    <CMD state="Updater" type="push" body="$ mkdir -p
/mnt/mmcblk%mmc%p1"/>
    <CMD state="Updater" type="push" body="$ mount -t vfat
/dev/mmcblk%mmc%p1 /mnt/mmcblk%mmc%p1"/>

    <!-- burn zImage -->
    <CMD state="Updater" type="push" body="send" file="files/zImage-
alientek-emmc">Sending kernel zImage</CMD>
    <CMD state="Updater" type="push" body="$ cp $FILE
/mnt/mmcblk%mmc%p1/zImage">write kernel image to sd card</CMD>

    <!-- burn dtb -->
    <CMD state="Updater" type="push" body="send" file="files/imx6ull-
alientek-emmc.dtb" ifdev="MX6ULL">Sending Device Tree file</CMD>
    <CMD state="Updater" type="push" body="$ cp $FILE
/mnt/mmcblk%mmc%p1/imx6ull-alientek-emmc.dtb" ifdev="MX6ULL">write
device tree to sd card</CMD>
    <CMD state="Updater" type="push" body="$ umount
/mnt/mmcblk%mmc%p1">Unmounting vfat partition</CMD>

    <!-- burn rootfs -->
    <CMD state="Updater" type="push" body="$ mkfs.ext3 -F -E nodiscard
/dev/mmcblk%mmc%p2">Formatting rootfs partition</CMD>
    <CMD state="Updater" type="push" body="$ mkdir -p
/mnt/mmcblk%mmc%p2"/>
    <CMD state="Updater" type="push" body="$ mount -t ext3
/dev/mmcblk%mmc%p2 /mnt/mmcblk%mmc%p2"/>
    <CMD state="Updater" type="push" body="pipe tar -jxv -C
/mnt/mmcblk%mmc%p2" file="files/rootfs-alientek-emmc.tar.bz2"
ifdev="MX6UL MX7D MX6ULL">Sending and writting rootfs</CMD>
    <CMD state="Updater" type="push" body="frf">Finishing rootfs
write</CMD>
    <CMD state="Updater" type="push" body="$ umount
/mnt/mmcblk%mmc%p2">Unmounting rootfs partition</CMD>
    <CMD state="Updater" type="push" body="$ echo Update
Complete!">Done</CMD>
```

```
  </LIST>
</UCL>
```

ucl2.xml 文件我们仅仅保留了给 EMMC 烧写系统，如果要支持 NAND 的话可以自行参考原版的 ucl2.xml 文件，添加相关的内容。

39.5.2 烧写测试

MfgTool 工具修改好以后就可以进行烧写测试了，将 imx6ull-alientek-emmc.dtb、u-boot-alientek-emmc.imx 和 zImage-alientek-emmc 这三个文件复制到 mfgtools-with-rootfs/mfgtools/Profiles/Linux/OS Firmware/firmware 目录中。将 imx6ull-alientek-emmc.dtb、u-boot-alientek-emmc.imx、zImage-alientek-emmc 和 rootfs-alientek-emmc.tar.bz2 这四个文件复制到 mfgtools-with-rootfs/mfgtools/Profiles/Linux/OS Firmware/files 目录中。

点击"mfgtool2-alientek-alpha-emmc.vbs"打开 MfgTool 烧写系统，等待烧写完成，然后设置拨码开关为 EMMC 启动，重启开发板，系统启动信息如图 39.5.2.1 所示：

```
Normal Boot
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc1(part 0) is current device
switch to partitions #0, OK
mmc1(part 0) is current device
reading boot.scr
** Unable to read file boot.scr **
reading zImage
5583352 bytes read in 138 ms (38.6 MiB/s)
Booting from mmc ...
reading imx6ull-14x14-evk.dtb
** Unable to read file imx6ull-14x14-evk.dtb **
Kernel image @ 0x80800000 [ 0x000000 - 0x5531f8 ]

Starting kernel ...
```

图 39.5.2.1 系统启动 log 信息

从图 39.5.2.1 可以看出，出现"Starting kernel ..."以后就再也没有任何信息输出了，说明 Linux 内核启动失败了。接下来就是解决为何 Linux 内核启动失败这个问题。

39.5.3 解决 Linux 内核启动失败

上一小节我们启动系统以后发现输出"Starting kernel ..."以后就再也没有任何信息了，难道是系统烧写错误了？可以确定的是 uboot 启动正常，就是在启动 Linux 的时候出问题了，仔细观察 uboot 输出的 log 信息，会发现如图 39.5.3.1 所示两行信息：

```
reading imx6ull-14x14-evk.dtb
** Unable to read file imx6ull-14x14-evk.dtb **
```

图 39.5.3.1 读取设备树出错

从图 39.5.3.1 可以看出，在读取"imx6ull-14x14-evk.dtb"这个设备树文件的时候出错了。重启 uboot，进入到命令行模式，输入如下命令查看 EMMC 的分区 1 里面有没有设备树文件：

```
mmc dev 1        //切换到 EMMC
ls mmc 1:1       //输出 EMMC1 分区 1 中的所有文件
```

结果如图 39.5.3.2 所示：

```
=>ls mmc 1:1
  5583352   zimage
    36185   imx6ull-alientek-emmc.dtb

2 file(s), 0 dir(s)

=>
```

图 39.5.3.2 EMMC 分区 1 文件

从图 39.5.3.2 可以看出，此时 EMMC 的分区 1 中是存在设备树文件的，只是文件名字为：imx6ull-alientek-emmc.dtb，因此读取 imx6ull-14x14-evk.dtb 肯定会出错的，因为根本就不存在这个文件。之所以出现这个错误的原因是因为 uboot 里面默认的设备树名字就是 imx6ull-14x14-evk.dtb，这个我们在讲解 uboot 的时候就已经说过了。解决方法很简单，有两种方法：

1、重新设置 bootcmd 环境变量值

进入 uboot 的命令行，重新设置 bootcmd 和 bootargs 这两个环境变量的值，这里要注意的是 bootargs 的值也要重新设置一下，命令如下：

```
setenv bootcmd 'mmc dev 1;fatload mmc 1:1 80800000 zImage;fatload mmc 1:1 83000000 imx6ull-alientek-emmc.dtb;bootz 80800000 - 83000000'
setenv bootargs 'console=ttymxc0,115200 root=/dev/mmcblk1p2 rootwait rw'
saveenv
```

设置好 bootcmd 和 bootargs 这两个环境变量以后重启开发板，Linux 系统就可以正常启动。

2、修改 uboot 源码

第 1 种方法每次重新烧写系统以后都要先手动设置一下 bootcmd 的值，这样有点麻烦，有没有一劳永逸的方法呢？肯定是有的，就是直接修改 uboot 源码。打开 uboot 源码中的文件 include/configs/mx6ull_alientek_emmc.h，在宏 CONFIG_EXTRA_ENV_SETTINGS 中找到如下所示内容：

示例代码 39.5.3.1 查找设备树文件

```
194 "findfdt="\
195  "if test $fdt_file = undefined; then " \
196     "if test $board_name = EVK && test $board_rev = 9X9; then " \
197         "setenv fdt_file imx6ull-9x9-evk.dtb; fi; " \
198     "if test $board_name = EVK && test $board_rev = 14X14; then " \
199         "setenv fdt_file imx6ull-14x14-evk.dtb; fi; " \
200     "if test $fdt_file = undefined; then " \
201         "echo WARNING: Could not determine dtb to use; fi; " \
202     "fi;\0" \
```

findfdt 就是用于确定设备树文件名字的环境变量，fdt_file 环境变量保存着设备树文件名。第 196 行和 197 行用于判断设备树文件名字是否为 imx6ull-9x9-evk.dtb，第 198 行和 199 行用于判断设备树文件名字是否为 imx6ull-14x14-evk.dtb。这两个设备树都是 NXP 官方开发板使用的，I.MX6U-ALPHA 开发板用不到，因此直接将示例代码 39.5.3.1 中 findfdt 的值改为如下内容：

示例代码 39.5.3.1 查找设备树文件

```
194 "findfdt="\
195    "if test $fdt_file = undefined; then " \
196        "setenv fdt_file imx6ull-alientek-emmc.dtb; " \
```

```
197     "fi;\0" \
```

第 196 行，如果 fdt_file 未定义的话，直接设置 fdt_file= imx6ull-alientek-emmc.dtb，简单直接，不需要任何的判断语句。修改后以后重新编译 uboot，然后用将新的 uboot 烧写到开发板中，烧写完成以后重启测试，Linux 内核启动正常。

关于系统烧写就讲解到这里，本章我们使用 NXP 提供的 MfgTool 工具通过 USB OTG 口向开发板的 EMMC 中烧写 uboot、Linux kernel、.dtb(设备树)和 rootfs 这四个文件。在本章我们主要做了五个工作：

①、理解 MfgTool 工具的工作原理。

②、使用 MfgTool 工具将 NXP 官方系统烧写到 I.MX6U-ALPHA 开发板中，主要是为了体验一下 MfgTool 软件的工作流程以及烧写方法。

③、使用 MfgTool 工具将我们自己编译出来的系统烧写到 I.MX6U-ALPHA 开发板中。

④、修改 MfgTool 工具，使其支持我们所使用的硬件平台。

⑤、修改相应的错误。

关于系统烧写的方法就讲解到这里，本章内容不仅仅是为了讲解如何向 I.MX6ULL 芯片中烧写系统，更重要的是向大家详细的讲解了 MfgTool 的工作原理。如果大家在后续的工作或学习中使用 I.MX7 或者 I.MX8 等芯片，本章同样适用。

随着本章的结束，也宣告着本书第三篇的内容也正式结束了，第三篇是系统移植篇，重点就是 uboot、Linux kernel 和 rootfs 的移植，看似简简单单的“移植”两个字，引出的却是一篇 300 多页的“爱恨情仇”。授人以鱼不如授人以渔，本可以简简单单的教大家修改哪些文件、添加哪些内容，怎么去编译，然后得到哪些文件。但是这样只能看到表象，并不能深入的了解其原理，为了让大家能够详细的了解整个流程，笔者义无反顾的选择了这条最难走的路，不管是 uboot 还是 Linux kernel，从 Makefile 到启动流程，都尽自己最大的努力去阐述清楚。奈何，笔者水平有限，还是有很多的细节没有处理好，大家有疑问的地方可以到正点原子论坛 www.openedv.com 上发帖留言，大家一起讨论学习。

第四篇 ARM Linux 驱动开发篇

前面 3 篇，我们学习 Ubuntu 操作系统、学习 ARM 裸机、学习系统移植，其目的就是为了本篇做准备。本篇应该是大家最期待的内容了，毕竟大部分学习者的最初目的就是学习 Linux 驱动开发。本篇我们将会详细讲解 Linux 中的三大类驱动：字符设备驱动、块设备驱动和网络设备驱动。其中字符设备驱动是占用篇幅最大的一类驱动，因为字符设备最多，从最简单的点灯到 I2C、SPI、音频等都属于字符设备驱动的类型。块设备和网络设备驱动要比字符设备驱动复杂，就是因为其复杂所以半导体厂商一般都给我们编写好了，大多数情况下都是直接可以使用的。所谓的块设备驱动就是存储器设备的驱动，比如 EMMC、NAND、SD 卡和 U 盘等存储设备，因为这些存储设备的特点是以存储块为基础，因此叫做块设备。网络设备驱动就更好理解了，就是网络驱动，不管是有线的还是无线的，都属于网络设备驱动的范畴。一个设备可以属于多种设备驱动类型，比如 USB WIFI，其使用 USB 接口，所以属于字符设备，但是其又能上网，所以也属于网络设备驱动。本篇我们就围绕着三大设备驱动类型展开，尽可能详细的讲解每种设备驱动的开发方式。

本书使用的 Linux 内核版本为 4.1.15，其支持设备树(Device tree)，所以本篇所有例程均采用设备树。设备树将是本篇的重点！从设备树的基本原理到设备树驱动的开发方式，从最简单的点灯到复杂的网络驱动开发，本篇均有详细的讲解，是学习设备树的不二之选。

最后，祝大家学习愉快！

第四十章 字符设备驱动开发

本章我们从 Linux 驱动开发中最基础的字符设备驱动开始，重点学习 Linux 下字符设备驱动开发框架。本章会以一个虚拟的设备为例，讲解如何进行字符设备驱动开发，以及如何编写测试 APP 来测试驱动工作是否正常，为以后的学习打下坚实的基础。

40.1 字符设备驱动简介

字符设备是 Linux 驱动中最基本的一类设备驱动，字符设备就是一个一个字节，按照字节流进行读写操作的设备，读写数据是分先后顺序的。比如我们最常见的点灯、按键、IIC、SPI，LCD 等等都是字符设备，这些设备的驱动就叫做字符设备驱动。

在详细的学习字符设备驱动架构之前，我们先来简单的了解一下 Linux 下的应用程序是如何调用驱动程序的，Linux 应用程序对驱动程序的调用如图 40.1.1 所示：

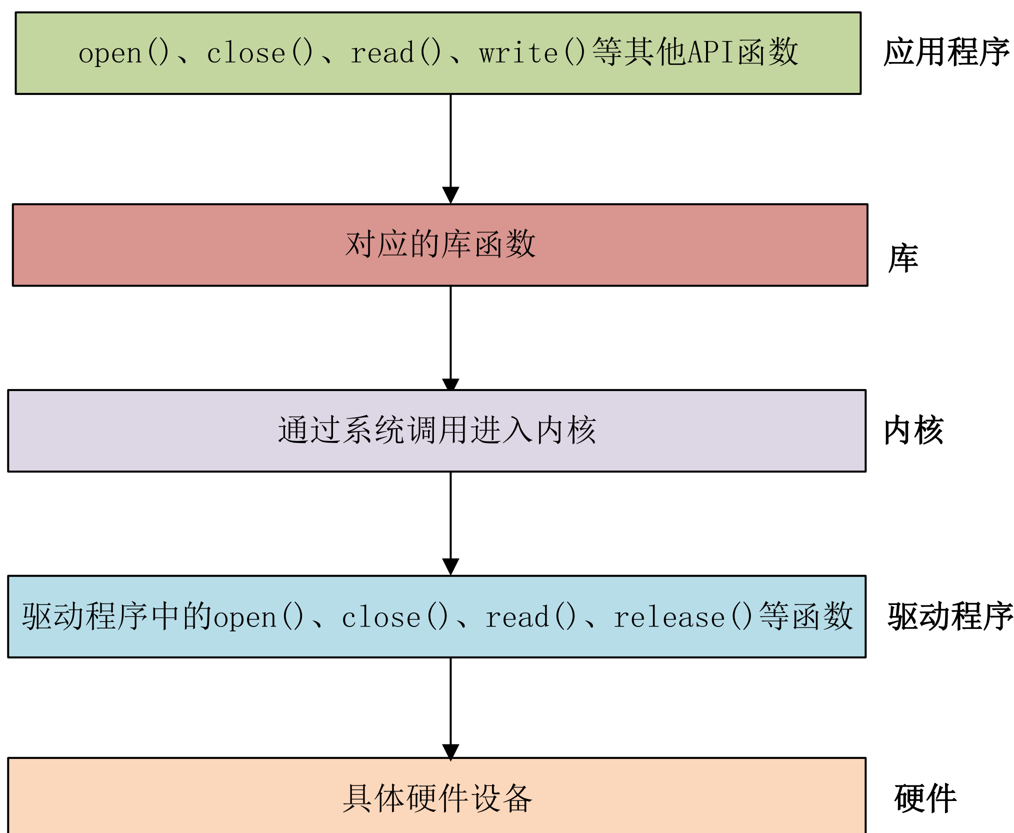


图 40.1.1 Linux 应用程序对驱动程序的调用流程

在 Linux 中一切皆为文件，驱动加载成功以后会在“/dev”目录下生成一个相应的文件，应用程序通过对这个名为“/dev/xxx” (xxx 是具体的驱动文件名字)的文件进行相应的操作即可实现对硬件的操作。比如现在有个叫做/dev/led 的驱动文件，此文件是 led 灯的驱动文件。应用程序使用 open 函数来打开文件/dev/led，使用完成以后使用 close 函数关闭/dev/led 这个文件。open 和 close 就是打开和关闭 led 驱动的函数，如果要点亮或关闭 led，那么就使用 write 函数来操作，也就是向此驱动写入数据，这个数据就是要关闭还是要打开 led 的控制参数。如果要获取 led 灯的状态，就用 read 函数从驱动中读取相应的状态。

应用程序运行在用户空间，而 Linux 驱动属于内核的一部分，因此驱动运行于内核空间。当我们在用户空间想要实现对内核的操作，比如使用 open 函数打开/dev/led 这个驱动，因为用户空间不能直接对内核进行操作，因此必须使用一个叫做“系统调用”的方法来实现从用户空间陷入到内核空间，这样才能实现对底层驱动的操作。open、close、write 和 read 等这些函数是有 C 库提供的，在 Linux 系统中，系统调用作为 C 库的一部分。当我们调用 open 函数的时候流程如图 40.1.2 所示：

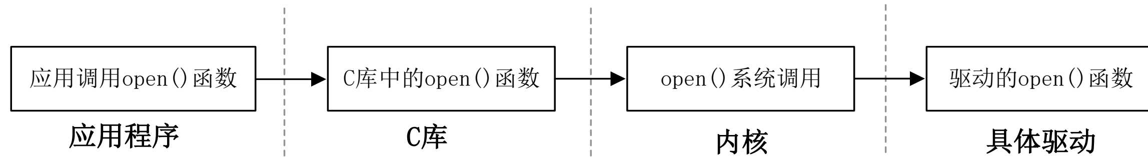


图 40.1.2 open 函数调用流程

其中关于 C 库以及如何通过系统调用陷入到内核空间这个我们不用去管，我们重点关注的是应用程序和具体的驱动，应用程序使用到的函数在具体驱动程序中都有与之对应的函数，比如应用程序中调用了 open 这个函数，那么在驱动程序中也得有一个名为 open 的函数。每一个系统调用，在驱动中都有与之对应的一个驱动函数，在 Linux 内核文件 include/linux/fs.h 中有个叫做 file_operations 的结构体，此结构体就是 Linux 内核驱动操作函数集合，内容如下所示：

示例代码 40.1.1 file_operations 结构体

```
1588 struct file_operations {
1589     struct module *owner;
1590     loff_t (*llseek) (struct file *, loff_t, int);
1591     ssize_t (*read) (struct file *, char __user *, size_t, loff_t
          *);
1592     ssize_t (*write) (struct file *, const char __user *, size_t,
          loff_t *);
1593     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
1594     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
1595     int (*iterate) (struct file *, struct dir_context *);
1596     unsigned int (*poll) (struct file *, struct poll_table_struct
          *);
1597     long (*unlocked_ioctl) (struct file *, unsigned int, unsigned
          long);
1598     long (*compat_ioctl) (struct file *, unsigned int, unsigned
          long);
1599     int (*mmap) (struct file *, struct vm_area_struct *);
1600     int (*mremap)(struct file *, struct vm_area_struct *);
1601     int (*open) (struct inode *, struct file *);
1602     int (*flush) (struct file *, fl_owner_t id);
1603     int (*release) (struct inode *, struct file *);
1604     int (*fsync) (struct file *, loff_t, loff_t, int datasync);
1605     int (*aio_fsync) (struct kiocb *, int datasync);
1606     int (*fasync) (int, struct file *, int);
1607     int (*lock) (struct file *, int, struct file_lock *);
1608     ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
          loff_t *, int);
1609     unsigned long (*get_unmapped_area)(struct file *, unsigned long,
          unsigned long, unsigned long, unsigned long);
1610     int (*check_flags)(int);
1611     int (*flock) (struct file *, int, struct file_lock *);
```

```
1612    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
         loff_t *, size_t, unsigned int);
1613    ssize_t (*splice_read)(struct file *, loff_t *, struct
         pipe_inode_info *, size_t, unsigned int);
1614    int (*setlease)(struct file *, long, struct file_lock **, void
         **);
1615    long (*fallocate)(struct file *file, int mode, loff_t offset,
1616              loff_t len);
1617    void (*show_fdinfo)(struct seq_file *m, struct file *f);
1618 #ifndef CONFIG_MMU
1619    unsigned (*mmap_capabilities)(struct file *);
1620 #endif
1621 };
```

简单介绍一下 file_operation 结构体中比较重要的、常用的函数：

第 1589 行，owner 拥有该结构体的模块的指针，一般设置为 THIS_MODULE。

第 1590 行，llseek 函数用于修改文件当前的读写位置。

第 1591 行，read 函数用于读取设备文件。

第 1592 行，write 函数用于向设备文件写入(发送)数据。

第 1596 行，poll 是个轮询函数，用于查询设备是否可以进行非阻塞的读写。

第 1597 行，unlocked_ioctl 函数提供对于设备的控制功能，与应用程序中的 ioctl 函数对应。

第 1598 行，compat_ioctl 函数与 unlocked_ioctl 函数功能一样，区别在于在 64 位系统上，32 位的应用程序调用将会使用此函数。在 32 位的系统上运行 32 位的应用程序调用的是 unlocked_ioctl。

第 1599 行，mmap 函数用于将将设备的内存映射到进程空间中(也就是用户空间)，一般帧缓冲设备会使用此函数，比如 LCD 驱动的显存，将帧缓冲(LCD 显存)映射到用户空间中以后应用程序就可以直接操作显存了，这样就不用在用户空间和内核空间之间来回复制。

第 1601 行，open 函数用于打开设备文件。

第 1603 行，release 函数用于释放(关闭)设备文件，与应用程序中的 close 函数对应。

第 1604 行，fasync 函数用于刷新待处理的数据，用于将缓冲区中的数据刷新到磁盘中。

第 1605 行，aio_fsync 函数与 fasync 函数的功能类似，只是 aio_fsync 是异步刷新待处理的数据。

在字符设备驱动开发中最常用的就是上面这些函数，关于其他的函数大家可以查阅相关文档。我们在字符设备驱动开发中最主要的工作就是实现上面这些函数，不一定全部都要实现，但是像 open、release、write、read 等都是需要实现的，当然了，具体需要实现哪些函数还是要看具体的驱动要求。

40.2 字符设备驱动开发步骤

上一小节我们简单的介绍了一下字符设备驱动，那么字符设备驱动开发都有哪些步骤呢？我们在学习裸机或者 STM32 的时候关于驱动的开发就是初始化相应的外设寄存器，在 Linux 驱动开发中肯定也是要初始化相应的外设寄存器，这个是毫无疑问的。只是在 Linux 驱动开发中我们需要按照其规定的框架来编写驱动，所以说学 Linux 驱动开发重点是学习其驱动框架。

40.2.1 驱动模块的加载和卸载

Linux 驱动有两种运行方式，第一种就是将驱动编译进 Linux 内核中，这样当 Linux 内核启动的时候就会自动运行驱动程序。第二种就是将驱动编译成模块(Linux 下模块扩展名为.ko)，在 Linux 内核启动以后使用“insmod”命令加载驱动模块。在调试驱动的时候一般都选择将其编译为模块，这样我们修改驱动以后只需要编译一下驱动代码即可，不需要编译整个 Linux 代码。而且在调试的时候只需要加载或者卸载驱动模块即可，不需要重启整个系统。总之，将驱动编译为模块最大的好处就是方便开发，当驱动开发完成，确定没有问题以后就可以将驱动编译进 Linux 内核中，当然也可以不编译进 Linux 内核中，具体看自己的需求。

模块有加载和卸载两种操作，我们在编写驱动的时候需要注册这两种操作函数，模块的加载和卸载注册函数如下：

```
module_init(xxx_init);        //注册模块加载函数
module_exit(xxx_exit);        //注册模块卸载函数
```

module_init 函数用来向 Linux 内核注册一个模块加载函数，参数 xxx_init 就是需要注册的具体函数，当使用“insmod”命令加载驱动的时候，xxx_init 这个函数就会被调用。module_exit() 函数用来向 Linux 内核注册一个模块卸载函数，参数 xxx_exit 就是需要注册的具体函数，当使用“rmmod”命令卸载具体驱动的时候 xxx_exit 函数就会被调用。字符设备驱动模块加载和卸载模板如下所示：

示例代码 40.2.1.1 字符设备驱动模块加载和卸载函数模板

```
1  /* 驱动入口函数 */
2  static int __init xxx_init(void)
3  {
4      /* 入口函数具体内容 */
5      return 0;
6  }
7
8  /* 驱动出口函数 */
9  static void __exit xxx_exit(void)
10 {
11     /* 出口函数具体内容 */
12 }
13
14 /* 将上面两个函数指定为驱动的入口和出口函数 */
15 module_init(xxx_init);
16 module_exit(xxx_exit);
```

第 2 行，定义了个名为 xxx_init 的驱动入口函数，并且使用了“__init”来修饰。

第 9 行，定义了个名为 xxx_exit 的驱动出口函数，并且使用了“__exit”来修饰。

第 15 行，调用函数 module_init 来声明 xxx_init 为驱动入口函数，当加载驱动的时候 xxx_init 函数就会被调用。

第 16 行，调用函数 module_exit 来声明 xxx_exit 为驱动出口函数，当卸载驱动的时候 xxx_exit 函数就会被调用。

驱动编译完成以后扩展名为.ko，有两种命令可以加载驱动模块：insmod 和 modprobe，insmod 是最简单的模块加载命令，此命令用于加载指定的.ko 模块，比如加载 drv.ko 这个驱动模块，命令如下：

```
insmod drv.ko
```

insmod 命令不能解决模块的依赖关系，比如 drv.ko 依赖 first.ko 这个模块，就必须先使用 insmod 命令加载 first.ko 这个模块，然后再加载 drv.ko 这个模块。但是 modprobe 就不会存在这个问题，modprobe 会分析模块的依赖关系，然后会将所有的依赖模块都加载到内核中，因此 modprobe 命令相比 insmod 要智能一些。modprobe 命令主要智能在提供了模块的依赖性分析、错误检查、错误报告等功能，推荐使用 modprobe 命令来加载驱动。modprobe 命令默认会去/lib/modules/<kernel-version>目录中查找模块，比如本书使用的 Linux kernel 的版本号为 4.1.15，因此 modprobe 命令默认会到/lib/modules/4.1.15 这个目录中查找相应的驱动模块，一般自己制作的根文件系统中是不会有这个目录的，所以需要自己手动创建。

驱动模块的卸载使用命令“rmmod”即可，比如要卸载 drv.ko，使用如下命令即可：

```
rmmod drv.ko
```

也可以使用“modprobe -r”命令卸载驱动，比如要卸载 drv.ko，命令如下：

```
modprobe -r drv.ko
```

使用 modprobe 命令可以卸载掉驱动模块所依赖的其他模块，前提是这些依赖模块已经没有被其他模块所使用，否则就不能使用 modprobe 来卸载驱动模块。所以对于模块的卸载，还是推荐使用 rmmod 命令。

40.2.2 字符设备注册与注销

对于字符设备驱动而言，当驱动模块加载成功以后需要注册字符设备，同样，卸载驱动模块的时候也需要注销掉字符设备。字符设备的注册和注销函数原型如下所示:

```
static inline int register_chrdev(unsigned int major, const char *name,
                                  const struct file_operations *fops)
static inline void unregister_chrdev(unsigned int major, const char *name)
```

register_chrdev 函数用于注册字符设备，此函数一共有三个参数，这三个参数的含义如下：

major：主设备号，Linux 下每个设备都有一个设备号，设备号分为主设备号和次设备号两部分，关于设备号后面会详细讲解。

name：设备名字，指向一串字符串。

fops：结构体 file_operations 类型指针，指向设备的操作函数集合变量。

unregister_chrdev 函数用户注销字符设备，此函数有两个参数，这两个参数含义如下：

major：要注销的设备对应的主设备号。

name：要注销的设备对应的设备名。

一般字符设备的注册在驱动模块的入口函数 xxx_init 中进行，字符设备的注销在驱动模块的出口函数 xxx_exit 中进行。在示例代码 40.2.1.1 中字符设备的注册和注销，内容如下所示：

示例代码 40.2.2.1 加入字符设备注册和注销

```
1 static struct file_operations test_fops;
2 
3 /* 驱动入口函数 */
4 static int __init xxx_init(void)
5 {
6     /* 入口函数具体内容 */
```

```
7       int retvalue = 0;
8
9       /* 注册字符设备驱动 */
10      retvalue = register_chrdev(200, "chrtest", &test_fops);
11      if(retvalue < 0){
12          /* 字符设备注册失败,自行处理 */
13      }
14      return 0;
15 }
16
17 /* 驱动出口函数 */
18 static void __exit xxx_exit(void)
19 {
20      /* 注销字符设备驱动 */
21      unregister_chrdev(200, "chrtest");
22 }
23
24 /* 将上面两个函数指定为驱动的入口和出口函数 */
25 module_init(xxx_init);
26 module_exit(xxx_exit);
```

第 1 行，定义了一个 file_operations 结构体变量 test_fops，test_fops 就是设备的操作函数集合，只是此时我们还没有初始化 test_fops 中的 open、release 等这些成员变量，所以这个操作函数集合还是空的。

第 10 行，调用函数 register_chrdev 注册字符设备，主设备号为 200，设备名字为“chrtest”，设备操作函数集合就是第 1 行定义的 test_fops。要注意的一点就是，选择没有被使用的主设备号，输入命令“cat /proc/devices”可以查看当前已经被使用掉的设备号，如图 40.2.2.1 所示(限于篇幅原因，只展示一部分)：

```
/ # cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 81 video4linux
```

图 40.2.2.1 查看当前设备

在图 40.2.2.1 中可以列出当前系统中所有的字符设备和块设备，其中第 1 列就是设备对应的主设备号。200 这个主设备号在我的开发板中并没有被使用，所以我这里就用了 200 这个主设备号。

第 21 行，调用函数 unregister_chrdev 注销主设备号为 200 的这个设备。

40.2.3 实现设备的具体操作函数

file_operations 结构体就是设备的具体操作函数，在示例代码 40.2.2.1 中我们定义了 file_operations 结构体类型的变量 test_fops，但是还没对其进行初始化，也就是初始化其中的 open、release、read 和 write 等具体的设备操作函数。本节小节我们就完成变量 test_fops 的初始化，设置好针对 chrtest 设备的操作函数。在初始化 test_fops 之前我们要分析一下需求，也就是要对 chrtest 这个设备进行哪些操作，只有确定了需求以后才知道我们应该实现哪些操作函数。假设对 chrtest 这个设备有如下两个要求：

1、能够对 chrtest 进行打开和关闭操作

设备打开和关闭是最基本的要求，几乎所有的设备都得提供打开和关闭的功能。因此我们需要实现 file_operations 中的 open 和 release 这两个函数。

2、对 chrtest 进行读写操作

假设 chrtest 这个设备控制着一段缓冲区(内存)，应用程序需要通过 read 和 write 这两个函数对 chrtest 的缓冲区进行读写操作。所以需要实现 file_operations 中的 read 和 write 这两个函数。

需求很清晰了，修改示例代码 40.2.2.1，在其中加入 test_fops 这个结构体变量的初始化操作，完成以后的内容如下所示：

示例代码 40.2.3.1 加入设备操作函数

```
1  /* 打开设备 */
2  static int chrtest_open(struct inode *inode, struct file *filp)
3  {
4      /* 用户实现具体功能 */
5      return 0;
6  }
7
8  /* 从设备读取 */
9  static ssize_t chrtest_read(struct file *filp, char __user *buf,
                                 size_t cnt, loff_t *offt)
10 {
11     /* 用户实现具体功能 */
12     return 0;
13 }
14
15 /* 向设备写数据 */
16 static ssize_t chrtest_write(struct file *filp,
                                 const char __user *buf,
                                 size_t cnt, loff_t *offt)
17 {
18     /* 用户实现具体功能 */
19     return 0;
20 }
21
```

```
22 /* 关闭/释放设备 */
23 static int chrtest_release(struct inode *inode, struct file *filp)
24 {
25     /* 用户实现具体功能 */
26     return 0;
27 }
28 
29 static struct file_operations test_fops = {
30     .owner = THIS_MODULE,
31     .open = chrtest_open,
32     .read = chrtest_read,
33     .write = chrtest_write,
34     .release = chrtest_release,
35 };
36 
37 /* 驱动入口函数 */
38 static int __init xxx_init(void)
39 {
40     /* 入口函数具体内容 */
41     int retvalue = 0;
42 
43     /* 注册字符设备驱动 */
44     retvalue = register_chrdev(200, "chrtest", &test_fops);
45     if(retvalue < 0){
46         /* 字符设备注册失败,自行处理 */
47     }
48     return 0;
49 }
50 
51 /* 驱动出口函数 */
52 static void __exit xxx_exit(void)
53 {
54     /* 注销字符设备驱动 */
55     unregister_chrdev(200, "chrtest");
56 }
57 
58 /* 将上面两个函数指定为驱动的入口和出口函数 */
59 module_init(xxx_init);
60 module_exit(xxx_exit);
```

在示例代码 40.2.3.1 中我们一开始编写了四个函数：chrtest_open、chrtest_read、chrtest_write 和 chrtest_release。这四个函数就是 chrtest 设备的 open、read、write 和 release 操作函数。第 29 行~35 行初始化 test_fops 的 open、read、write 和 release 这四个成员变量。

40.2.4 添加 LICENSE 和作者信息

最后我们需要在驱动中加入 LICENSE 信息和作者信息，其中 LICENSE 是必须添加的，否则的话编译的时候会报错，作者信息可以添加也可以不添加。LICENSE 和作者信息的添加使用如下两个函数：

```
MODULE_LICENSE()        //添加模块 LICENSE 信息
MODULE_AUTHOR()         //添加模块作者信息
```

最后给示例代码 40.2.3.1 加入 LICENSE 和作者信息，完成以后的内容如下：

示例代码 40.2.4.1 字符设备驱动最终的模板

```
1  /* 打开设备 */
2  static int chrtest_open(struct inode *inode, struct file *filp)
3  {
4      /* 用户实现具体功能 */
5      return 0;
6  }
......
57
58 /* 将上面两个函数指定为驱动的入口和出口函数 */
59 module_init(xxx_init);
60 module_exit(xxx_exit);
61
62 MODULE_LICENSE("GPL");
63 MODULE_AUTHOR("zuozhongkai");
```

第 62 行，LICENSE 采用 GPL 协议。

第 63 行，添加作者名字。

至此，字符设备驱动开发的完整步骤就讲解完了，而且也编写好了一个完整的字符设备驱动模板，以后字符设备驱动开发都可以在此模板上进行。

40.3 Linux 设备号

40.3.1 设备号的组成

为了方便管理，Linux 中每个设备都有一个设备号，设备号由主设备号和次设备号两部分组成，主设备号表示某一个具体的驱动，次设备号表示使用这个驱动的各个设备。Linux 提供了一个名为 dev_t 的数据类型表示设备号，dev_t 定义在文件 include/linux/types.h 里面，定义如下：

示例代码 40.3.1 设备号 dev_t

```
12 typedef __u32 __kernel_dev_t;
......
15 typedef __kernel_dev_t dev_t;
```

可以看出 dev_t 是__u32 类型的，而__u32 定义在文件 include/uapi/asm-generic/int-ll64.h 里面，定义如下：

示例代码 40.3.2 __u32 类型

```
26 typedef unsigned int __u32;
```

综上所述，dev_t 其实就是 unsigned int 类型，是一个 32 位的数据类型。这 32 位的数据构

成了主设备号和次设备号两部分，其中高 12 位为主设备号，第 20 位为次设备号。因此 Linux 系统中主设备号范围为 0~4095，所以大家在选择主设备号的时候一定不要超过这个范围。在文件 include/linux/kdev_t.h 中提供了几个关于设备号的操作函数(本质是宏)，如下所示：

示例代码 40.3.3 设备号操作函数

```
6  #define MINORBITS    20
7  #define MINORMASK    ((1U << MINORBITS) - 1)
8
9  #define MAJOR(dev)   ((unsigned int) ((dev) >> MINORBITS))
10 #define MINOR(dev)   ((unsigned int) ((dev) & MINORMASK))
11 #define MKDEV(ma,mi) (((ma) << MINORBITS) | (mi))
```

第 6 行，宏 MINORBITS 表示次设备号位数，一共是 20 位。

第 7 行，宏 MINORMASK 表示次设备号掩码。

第 9 行，宏 MAJOR 用于从 dev_t 中获取主设备号，将 dev_t 右移 20 位即可。

第 10 行，宏 MINOR 用于从 dev_t 中获取此设备号，取 dev_t 的低 20 位的值即可。

第 11 行，宏 MKDEV 用于将给定的主设备号和次设备号的值组合成 dev_t 类型的设备号。

40.3.2 设备号的分配

1、静态分配设备号

本小节讲的设备号分配主要是主设备号的分配。前面讲解字符设备驱动的时候说过了，注册字符设备的时候需要给设备指定一个设备号，这个设备号可以是驱动开发者静态的指定一个设备号，比如选择 200 这个主设备号。有一些常用的设备号已经被 Linux 内核开发者给分配掉了，具体分配的内容可以查看文档 Documentation/devices.txt。并不是说内核开发者已经分配掉的主设备号我们就不能用了，具体能不能用还得看我们的硬件平台运行过程中有没有使用这个主设备号，使用“cat /proc/devices”命令即可查看当前系统中所有已经使用了的设备号。

2、动态分配设备号

静态分配设备号需要我们检查当前系统中所有被使用了的设备号，然后挑选一个没有使用的。而且静态分配设备号很容易带来冲突问题，Linux 社区推荐使用动态分配设备号，在注册字符设备之前先申请一个设备号，系统会自动给你一个没有被使用的设备号，这样就避免了冲突。卸载驱动的时候释放掉这个设备号即可，设备号的申请函数如下：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

函数 alloc_chrdev_region 用于申请设备号，此函数有 4 个参数：

dev：保存申请到的设备号。

baseminor：次设备号起始地址，alloc_chrdev_region 可以申请一段连续的多个设备号，这些设备号的主设备号一样，但是次设备号不同，次设备号以 baseminor 为起始地址地址开始递增。一般 baseminor 为 0，也就是说次设备号从 0 开始。

count：要申请的设备号数量。

name：设备名字。

注销字符设备之后要释放掉设备号，设备号释放函数如下：

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

此函数有两个参数：

from：要释放的设备号。

count：表示从 from 开始，要释放的设备号数量。

40.4 chrdevbase 字符设备驱动开发实验

字符设备驱动开发的基本步骤我们已经了解了，本节我们就以 chrdevbase 这个虚拟设备为例，完整的编写一个字符设备驱动模块。chrdevbase 不是实际存在的一个设备，是笔者为了方便讲解字符设备的开发而引入的一个虚拟设备。chrdevbase 设备有两个缓冲区，一个为读缓冲区，一个为写缓冲区，这两个缓冲区的大小都为 100 字节。在应用程序中可以向 chrdevbase 设备的写缓冲区中写入数据，从读缓冲区中读取数据。chrdevbase 这个虚拟设备的功能很简单，但是它包含了字符设备的最基本功能。

40.4.1 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 1_chrdevbase。

应用程序调用 open 函数打开 chrdevbase 这个设备，打开以后可以使用 write 函数向 chrdevbase 的写缓冲区 writebuf 中写入数据(不超过 100 个字节)，也可以使用 read 函数读取读缓冲区 readbuf 中的数据操作，操作完成以后应用程序使用 close 函数关闭 chrdevbase 设备。

1、创建 VSCode 工程

在 Ubuntu 中创建一个目录用来存放 Linux 驱动程序，比如我创建了一个名为 Linux_Drivers 的目录来存放所有的 Linux 驱动。在 Linux_Drivers 目录下新建一个名为 1_chrdevbase 的子目录来存放本实验所有文件，如图 40.4.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers$ ls
1_chrdevbase
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers$
```

图 40.4.1.1 Linux 实验程序目录

在 1_chrdevbase 目录中新建 VSCode 工程，并且新建 chrdevbase.c 文件，完成以后 1_chrdevbase 目录中的文件如图 40.4.1.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$ ls -a
.  ..  1_chrdevbase.code-workspace  chrdevbase.c  .dist  .vscode
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$
```

图 40.4.1.2 1_chardevbase 目录文件

2、添加头文件路径

因为是编写 Linux 驱动，因此会用到 Linux 源码中的函数。我们需要在 VSCode 中添加 Linux 源码中的头文件路径。打开 VSCode，按下“Crtl+Shift+P”打开 VSCode 的控制台，然后输入“C/C++: Edit configurations(JSON) ”，打开 C/C++编辑配置文件，如图 40.4.1.3 所示：

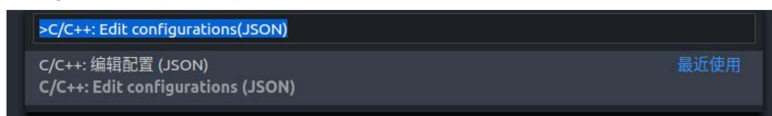


图 40.4.1.3 C/C++编辑配置文件。

打开以后会自动在.vscode 目录下生成一个名为 c_cpp_properties.json 的文件，此文件默认内容如下所示：

示例代码 40.4.1.1 c_cpp_properties.json 文件原内容

```
1 {
2     "configurations": [
3         {
4             "name": "Linux",
```

```
5               "includePath": [
6                   "${workspaceFolder}/**",
7                   ],
8               "defines": [],
9               "compilerPath": "/usr/bin/clang",
10              "cStandard": "c11",
11              "cppStandard": "c++17",
12              "intelliSenseMode": "clang-x64"
13          }
14      ],
15      "version": 4
16  }
```

第 5 行的 includePath 表示头文件路径，需要将 Linux 源码里面的头文件路径添加进来，也就是我们前面移植的 Linux 源码中的头文件路径。添加头文件路径以后的 c_cpp_properties.json 的文件内容如下所示：

示例代码 40.4.1.2 添加头文件路径后的 c_cpp_properties.json

```
1  {
2      "configurations": [
3          {
4              "name": "Linux",
5              "includePath": [
6                  "${workspaceFolder}/**",
7                  "/home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                    rel_imx_4.1.15_2.1.0_ga_alientek/include",
8                  "/home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                    rel_imx_4.1.15_2.1.0_ga_alientek/arch/arm/include",
9                  "/home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                    rel_imx_4.1.15_2.1.0_ga_alientek/arch/arm/include/gener
                    ated/"
10             ],
11             "defines": [],
......
16         }
17     ],
18     "version": 4
19 }
```

第 7~9 行就是添加好的 Linux 头文件路径。分别是开发板所使用的 Linux 源码下的 include、arch/arm/include 和 arch/arm/include/generated 这三个目录的路径，注意，这里使用了绝对路径。

3、编写实验程序

工程建立好以后就可以开始编写驱动程序了，新建 chrdevbase.c，然后在里面输入如下内容：

示例代码 40.4.1.3 chrdevbase.c 文件

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
```

```
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
7   /***************************************************************
8   Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
9   文件名   : chrdevbase.c
10  作者     : 左忠凯
11  版本     : V1.0
12  描述     : chrdevbase 驱动文件。
13  其他     : 无
14  论坛     : www.openedv.com
15  日志     : 初版 V1.0 2019/1/30 左忠凯创建
16  ***************************************************************/
17
18  #define CHRDEVBASE_MAJOR    200             /* 主设备号     */
19  #define CHRDEVBASE_NAME     "chrdevbase"    /* 设备名       */
20
21  static char readbuf[100];                   /* 读缓冲区     */
22  static char writebuf[100];                  /* 写缓冲区     */
23  static char kerneldata[] = {"kernel data!"};
24
25  /*
26   * @description   : 打开设备
27   * @param – inode : 传递给驱动的 inode
28   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
29   *                 一般在 open 的时候将 private_data 指向设备结构体。
30   * @return        : 0 成功;其他 失败
31   */
32  static int chrdevbase_open(struct inode *inode, struct file *filp)
33  {
34      //printk("chrdevbase open!\r\n");
35      return 0;
36  }
37
38  /*
39   * @description   : 从设备读取数据
40   * @param - filp  : 要打开的设备文件(文件描述符)
41   * @param - buf   : 返回给用户空间的数据缓冲区
42   * @param - cnt   : 要读取的数据长度
43   * @param - offt  : 相对于文件首地址的偏移
44   * @return        : 读取的字节数，如果为负值，表示读取失败
45   */
```

```
46 static ssize_t chrdevbase_read(struct file *filp, char __user *buf,
                                   size_t cnt, loff_t *offt)
47 {
48     int retvalue = 0;
49 
50     /* 向用户空间发送数据 */
51     memcpy(readbuf, kerneldata, sizeof(kerneldata));
52     retvalue = copy_to_user(buf, readbuf, cnt);
53     if(retvalue == 0){
54         printk("kernel senddata ok!\r\n");
55     }else{
56         printk("kernel senddata failed!\r\n");
57     }
58 
59     //printk("chrdevbase read!\r\n");
60     return 0;
61 }
62 
63 /*
64  * @description   : 向设备写数据
65  * @param - filp  : 设备文件，表示打开的文件描述符
66  * @param - buf   : 要写给设备写入的数据
67  * @param - cnt   : 要写入的数据长度
68  * @param - offt  : 相对于文件首地址的偏移
69  * @return        : 写入的字节数，如果为负值，表示写入失败
70  */
71 static ssize_t chrdevbase_write(struct file *filp,
                                    const char __user *buf,
                                    size_t cnt, loff_t *offt)
72 {
73     int retvalue = 0;
74     /* 接收用户空间传递给内核的数据并且打印出来 */
75     retvalue = copy_from_user(writebuf, buf, cnt);
76     if(retvalue == 0){
77         printk("kernel recevdata:%s\r\n", writebuf);
78     }else{
79         printk("kernel recevdata failed!\r\n");
80     }
81 
82     //printk("chrdevbase write!\r\n");
83     return 0;
84 }
85 
```

```
86  /*
87   * @description   : 关闭/释放设备
88   * @param - filp  : 要关闭的设备文件(文件描述符)
89   * @return        : 0 成功;其他 失败
90   */
91  static int chrdevbase_release(struct inode *inode,
                                   struct file *filp)
92  {
93      //printk("chrdevbase release! \r\n");
94      return 0;
95  }
96
97  /*
98   * 设备操作函数结构体
99   */
100 static struct file_operations chrdevbase_fops = {
101     .owner = THIS_MODULE,
102     .open = chrdevbase_open,
103     .read = chrdevbase_read,
104     .write = chrdevbase_write,
105     .release = chrdevbase_release,
106 };
107
108 /*
109  * @description   : 驱动入口函数
110  * @param         : 无
111  * @return        : 0 成功;其他 失败
112  */
113 static int __init chrdevbase_init(void)
114 {
115     int retvalue = 0;
116
117     /* 注册字符设备驱动 */
118     retvalue = register_chrdev(CHRDEVBASE_MAJOR, CHRDEVBASE_NAME,
                                    &chrdevbase_fops);
119     if(retvalue < 0){
120         printk("chrdevbase driver register failed\r\n");
121     }
122     printk("chrdevbase_init()\r\n");
123     return 0;
124 }
125
126 /*
```

```
127  * @description  : 驱动出口函数
128  * @param        : 无
129  * @return       : 无
130  */
131 static void __exit chrdevbase_exit(void)
132 {
133     /* 注销字符设备驱动 */
134     unregister_chrdev(CHRDEVBASE_MAJOR, CHRDEVBASE_NAME);
135     printk("chrdevbase_exit()\r\n");
136 }
137
138 /*
139  * 将上面两个函数指定为驱动的入口和出口函数
140  */
141 module_init(chrdevbase_init);
142 module_exit(chrdevbase_exit);
143
144 /*
145  * LICENSE 和作者信息
146  */
147 MODULE_LICENSE("GPL");
148 MODULE_AUTHOR("zuozhongkai");
```

第 32~36 行，chrdevbase_open 函数，当应用程序调用 open 函数的时候此函数就会调用，本例程中我们没有做任何工作，只是输出一串字符，用于调试。这里使用了 printk 来输出信息，而不是 printf！因为在 Linux 内核中没有 printf 这个函数。printk 相当于 printf 的孪生兄妹，printf 运行在用户态，printk 运行在内核态。在内核中想要向控制台输出或显示一些内容，必须使用 printk 这个函数。不同之处在于，printk 可以根据日志级别对消息进行分类，一共有 8 个消息级别，这 8 个消息级别定义在文件 include/linux/kern_levels.h 里面，定义如下：

```
#define KERN_SOH        "\001"
#define KERN_EMERG      KERN_SOH "0"   /* 紧急事件，一般是内核崩溃           */
#define KERN_ALERT      KERN_SOH "1"   /* 必须立即采取行动                   */
#define KERN_CRIT       KERN_SOH "2"   /* 临界条件，比如严重的软件或硬件错误*/
#define KERN_ERR        KERN_SOH "3"   /* 错误状态，一般设备驱动程序中使用
                                          KERN_ERR 报告硬件错误              */
#define KERN_WARNING    KERN_SOH "4"   /* 警告信息，不会对系统造成严重影响   */
#define KERN_NOTICE     KERN_SOH "5"   /* 有必要进行提示的一些信息           */
#define KERN_INFO       KERN_SOH "6"   /* 提示性的信息                       */
#define KERN_DEBUG      KERN_SOH "7"   /* 调试信息                           */
```

一共定义了 8 个级别，其中 0 的优先级最高，7 的优先级最低。如果要设置消息级别，参考如下示例：

```
printk(KERN_EMERG "gsmi: Log Shutdown Reason\n");
```

上述代码就是设置“gsmi: Log Shutdown Reason\n”这行消息的级别为 KERN_EMERG。在具体的消息前面加上 KERN_EMERG 就可以将这条消息的级别设置为 KERN_EMERG。如果使

用 printk 的时候不显式的设置消息级别，那么 printk 将会采用默认级别 MESSAGE_LOGLEVEL_DEFAULT，MESSAGE_LOGLEVEL_DEFAULT 默认为 4。

在 include/linux/printk.h 中有个宏 CONSOLE_LOGLEVEL_DEFAULT，定义如下：

```
#define CONSOLE_LOGLEVEL_DEFAULT 7
```

CONSOLE_LOGLEVEL_DEFAULT 控制着哪些级别的消息可以显示在控制台上，此宏默认为 7，意味着只有优先级高于 7 的消息才能显示在控制台上。

这个就是 printk 和 printf 的最大区别，可以通过消息级别来决定哪些消息可以显示在控制台上。默认消息级别为 4，4 的级别比 7 高，所示直接使用 printk 输出的信息是可以显示在控制台上的。

参数 filp 有个叫做 private_data 的成员变量，private_data 是个 void 指针，一般在驱动中将 private_data 指向设备结构体，设备结构体会存放设备的一些属性。

第 46~61 行，chrdevbase_read 函数，应用程序调用 read 函数从设备中读取数据的时候此函数会执行。参数 buf 是用户空间的内存，读取到的数据存储在 buf 中，参数 cnt 是要读取的字节数，参数 offt 是相对于文件首地址的偏移。kerneldata 里面保存着用户空间要读取的数据，第 51 行先将 kerneldata 数组中的数据拷贝到读缓冲区 readbuf 中，第 52 行通过函数 copy_to_user 将 readbuf 中的数据复制到参数 buf 中。因为内核空间不能直接操作用户空间的内存，因此需要借助 copy_to_user 函数来完成内核空间的数据到用户空间的复制。copy_to_user 函数原型如下：

```
static inline long copy_to_user(void __user *to, const void *from, unsigned long n)
```

参数 to 表示目的，参数 from 表示源，参数 n 表示要复制的数据长度。如果复制成功，返回值为 0，如果复制失败则返回负数。

第 71~84 行，chrdevbase_write 函数，应用程序调用 write 函数向设备写数据的时候此函数就会执行。参数 buf 就是应用程序要写入设备的数据，也是用户空间的内存，参数 cnt 是要写入的数据长度，参数 offt 是相对文件首地址的偏移。第 75 行通过函数 copy_from_user 将 buf 中的数据复制到写缓冲区 writebuf 中，因为用户空间内存不能直接访问内核空间的内存，所以需要借助函数 copy_from_user 将用户空间的数据复制到 writebuf 这个内核空间中。

第 91~95 行，chrdevbase_release 函数，应用程序调用 close 关闭设备文件的时候此函数会执行，一般会在此函数里面执行一些释放操作。如果在 open 函数中设置了 filp 的 private_data 成员变量指向设备结构体，那么在 release 函数最终就要释放掉。

第 100~106 行，新建 chrdevbase 的设备文件操作结构体 chrdevbase_fops，初始化 chrdevbase_fops。

第 113~124 行，驱动入口函数 chrdevbase_init，第 118 行调用函数 register_chrdev 来注册字符设备。

第 131~136 行，驱动出口函数 chrdevbase_exit，第 134 行调用函数 unregister_chrdev 来注销字符设备。

第 141~142 行，通过 module_init 和 module_exit 这两个函数来指定驱动的入口和出口函数。

第 147~148 行，添加 LICENSE 和作者信息。

40.4.2 编写测试 APP

1、C 库文件操作基本函数

编写测试 APP 就是编写 Linux 应用，需要用到 C 库里面和文件操作有关的一些函数，比如 open、read、write 和 close 这四个函数。

①、open 函数

open 函数原型如下：

```
int open(const char *pathname, int flags)
```

open 函数参数含义如下：

pathname：要打开的设备或者文件名。

flags：文件打开模式，以下三种模式必选其一：

O_RDONLY	只读模式
O_WRONLY	只写模式
O_RDWR	读写模式

因为我们要对 chrdevbase 这个设备进行读写操作，所以选择 O_RDWR。除了上述三种模式以外还有其他的可选模式，通过逻辑或来选择多种模式：

O_APPEND	每次写操作都写入文件的末尾
O_CREAT	如果指定文件不存在，则创建这个文件
O_EXCL	如果要创建的文件已存在，则返回 -1，并且修改 errno 的值
O_TRUNC	如果文件存在，并且以只写/读写方式打开，则清空文件全部内容
O_NOCTTY	如果路径名指向终端设备，不要把这个设备用作控制终端。
O_NONBLOCK	如果路径名指向 FIFO/块文件/字符文件，则把文件的打开和后继 I/O 设置为非阻塞
DSYNC	等待物理 I/O 结束后再 write。在不影响读取新写入的数据的前提下，不等待文件属性更新。
O_RSYNC	read 等待所有写入同一区域的写操作完成后再进行。
O_SYNC	等待物理 I/O 结束后再 write，包括更新文件属性的 I/O。

返回值：如果文件打开成功的话返回文件的文件描述符。

在 Ubuntu 中输入“man 2 open”即可查看 open 函数的详细内容，如图 40.4.2.1 所示：

```
zuozhongkai@ubuntu: ~
OPEN(2)                                          Linux Programmer's Manual
                             OPEN(2)

NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

       int openat(int dirfd, const char *pathname, int flags);
       int openat(int dirfd, const char *pathname, int flags, mode_t mode);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       openat():
           Since glibc 2.10:
 Manual page open(2) line 1 (press h for help or q to quit)
```

图 40.4.2.1 open 函数帮助信息

②、read 函数

read 函数原型如下：

```
ssize_t read(int fd, void *buf, size_t count)
```

read 函数参数含义如下：

fd：要读取的文件描述符，读取文件之前要先用 open 函数打开文件，open 函数打开文件成功以后会得到文件描述符。

buf：数据读取到此 buf 中。

count：要读取的数据长度，也就是字节数。

返回值：读取成功的话返回读取到的字节数；如果返回 0 表示读取到了文件末尾；如果返回负值，表示读取失败。在 Ubuntu 中输入“man 2 read”命令即可查看 read 函数的详细内容。

③、write 函数

write 函数原型如下：

```
ssize_t write(int fd, const void *buf, size_t count);
```

write 函数参数含义如下：

fd：要进行写操作的文件描述符，写文件之前要先用 open 函数打开文件，open 函数打开文件成功以后会得到文件描述符。

buf：要写入的数据。

count：要写入的数据长度，也就是字节数。

返回值：写入成功的话返回写入的字节数；如果返回 0 表示没有写入任何数据；如果返回负值，表示写入失败。在 Ubuntu 中输入“man 2 write”命令即可查看 write 函数的详细内容。

④、close 函数

close 函数原型如下：

```
int close(int fd);
```

close 函数参数含义如下：

fd：要关闭的文件描述符。

返回值：0 表示关闭成功，负值表示关闭失败。在 Ubuntu 中输入“man 2 close”命令即可查看 close 函数的详细内容。

2、编写测试 APP 程序

驱动编写好以后是需要测试的，一般编写一个简单的测试 APP，测试 APP 运行在用户空间。测试 APP 很简单通过输入相应的指令来对 chrdevbase 设备执行读或者写操作。在 1_chrdevbase 目录中新建 chrdevbaseApp.c 文件，在此文件中输入如下内容：

示例代码 40.4.2.1 chrdevbaseApp.c 文件

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名      : chrdevbaseApp.c
11 作者        : 左忠凯
12 版本        : V1.0
13 描述        : chrdevbase 驱测试 APP。
14 其他        : 使用方法：./chrdevbaseApp /dev/chrdevbase <1>|<2>
```

```
15             argv[2] 1:读文件
16             argv[2] 2:写文件
17 论坛        : www.openedv.com
18 日志        : 初版 V1.0 2019/1/30 左忠凯创建
19 ***************************************************************/
20
21 static char usrdata[] = {"usr data!"};
22
23 /*
24  * @description     : main 主程序
25  * @param - argc    : argv 数组元素个数
26  * @param - argv    : 具体参数
27  * @return          : 0 成功;其他 失败
28  */
29 int main(int argc, char *argv[])
30 {
31     int fd, retvalue;
32     char *filename;
33     char readbuf[100], writebuf[100];
34
35     if(argc != 3){
36         printf("Error Usage!\r\n");
37         return -1;
38     }
39
40     filename = argv[1];
41
42     /* 打开驱动文件 */
43     fd  = open(filename, O_RDWR);
44     if(fd < 0){
45         printf("Can't open file %s\r\n", filename);
46         return -1;
47     }
48
49     if(atoi(argv[2]) == 1){ /* 从驱动文件读取数据 */
50         retvalue = read(fd, readbuf, 50);
51         if(retvalue < 0){
52             printf("read file %s failed!\r\n", filename);
53         }else{
54             /*  读取成功，打印出读取成功的数据 */
55             printf("read data:%s\r\n",readbuf);
56         }
57     }
```

```
58
59      if(atoi(argv[2]) == 2){
60          /* 向设备驱动写数据 */
61          memcpy(writebuf, usrdata, sizeof(usrdata));
62          retvalue = write(fd, writebuf, 50);
63          if(retvalue < 0){
64              printf("write file %s failed!\r\n", filename);
65          }
66      }
67
68      /* 关闭设备 */
69      retvalue = close(fd);
70      if(retvalue < 0){
71          printf("Can't close file %s\r\n", filename);
72          return -1;
73      }
74
75      return 0;
76  }
```

第 21 行，数组 usrdata 是测试 APP 要向 chrdevbase 设备写入的数据。

第 35 行，判断运行测试 APP 的时候输入的参数是不是为 3 个，main 函数的 argc 参数表示参数数量，argv[]保存着具体的参数，如果参数不为 3 个的话就表示测试 APP 用法错误。比如，现在要从 chrdevbase 设备中读取数据，需要输入如下命令：

```
./chrdevbaseApp /dev/chrdevbase 1
```

上述命令一共有三个参数“./chrdevbaseApp”、“/dev/chrdevbase”和“1”，这三个参数分别对应 argv[0]、argv[1]和 argv[2]。第一个参数表示运行 chrdevbaseAPP 这个软件，第二个参数表示测试 APP 要打开/dev/chrdevbase 这个设备。第三个参数就是要执行的操作，1 表示从 chrdevbase 中读取数据，2 表示向 chrdevbase 写数据。

第 40 行，获取要打开的设备文件名字，argv[1]保存着设备名字。

第 43 行，调用 C 库中的 open 函数打开设备文件：/dev/chrdevbase。

第 49 行，判断 argv[2]参数的值是 1 还是 2，因为输入命令的时候其参数都是字符串格式的，因此需要借助 atoi 函数将字符串格式的数字转换为真实的数字。

第 50 行，当 argv[2]为 1 的时候表示要从 chrdevbase 设备中读取数据，一共读取 50 字节的数据，读取到的数据保存在 readbuf 中，读取成功以后就在终端上打印出读取到的数据。

第 59 行，当 argv[2]为 2 的时候表示要向 chrdevbase 设备写数据。

第 69 行，对 chrdevbase 设备操作完成以后就关闭设备。

chrdevbaseApp.c 内容还是很简单的，就是最普通的文件打开、关闭和读写操作。

40.4.3 编译驱动程序和测试 APP

1、编译驱动程序

首先编译驱动程序，也就是 chrdevbase.c 这个文件，我们需要将其编译为.ko 模块，创建 Makefile 文件，然后在其中输入如下内容：

示例代码 40.4.3.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                 rel_imx_4.1.15_2.1.0_ga_alientek
2  CURRENT_PATH := $(shell pwd)
3  obj-m := chrdevbase.o
4
5  build: kernel_modules
6
7  kernel_modules:
8      $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) modules
9  clean:
10     $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 1 行，KERNELDIR 表示开发板所使用的 Linux 内核源码目录，使用绝对路径，大家根据自己的实际情况填写即可。

第 2 行，CURRENT_PATH 表示当前路径，直接通过运行“pwd”命令来获取当前所处路径。

第 3 行，obj-m 表示将 chrdevbase.c 这个文件，并且编译为模块。

第 8 行，具体的编译命令，后面的 modules 表示编译模块，-C 表示将当前的工作目录切换到指定目录中，也就是 KERNERLDIR 目录。M 表示模块源码目录，“make modules”命令中加入 M=dir 以后程序会自动到指定的 dir 目录中读取模块的源码并将其编译为.ko 文件。

Makefile 编写好以后输入“make”命令编译驱动模块，编译过程如图 40.4.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$ ls
1_chrdevbase.code-workspace  chrdevbaseApp.c  chrdevbase.c  Makefile
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$ make -j32
make -C /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek M=/home/zuozhongkai/li
nux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase modules
make[1]: Entering directory '/home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek'
  CC [M]  /home/zuozhongkai/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase/chrdevbase.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/zuozhongkai/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase/chrdevbase.mod.o
  LD [M]  /home/zuozhongkai/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase/chrdevbase.ko
make[1]: Leaving directory '/home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek'
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$
```

图 40.4.3.1 驱动模块编译过程

编译成功以后就会生成一个叫做 chrdevbaes.ko 的文件，此文件就是 chrdevbase 设备的驱动模块。至此，chrdevbase 设备的驱动就编译成功。

2、编译测试 APP

测试 APP 比较简单，只有一个文件，因此就不需要编写 Makefile 了，直接输入命令编译。因为测试 APP 是要在 ARM 开发板上运行的，所以需要使用 arm-linux-gnueabihf-gcc 来编译，输入如下命令：

```
arm-linux-gnueabihf-gcc chrdevbaseApp.c -o chrdevbaseApp
```

编译完成以后会生成一个叫做 chrdevbaseApp 的可执行程序，输入如下命令查看 chrdevbaseAPP 这个程序的文件信息：

```
file chrdevbaseApp
```

结果如图 40.4.3.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$ file chrdevbaseApp
chrdevbaseApp: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpr
eter /lib/ld-, for GNU/Linux 2.6.31, BuildID[sha1]=5d017375992cf6c40e8fccb19a238dfd552ca7b6, not s
tripped
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/1_chrdevbase$
```

图 40.4.3.2 chrdevbaseAPP 文件信息

从图 40.4.3.2 可以看出，chrdevbaseAPP 这个可执行文件是 32 位 LSB 格式，ARM 版本的，因此 chrdevbaseAPP 只能在 ARM 芯片下运行。

40.4.4 运行测试

1、加载驱动模块

驱动模块 chrdevbase.ko 和测试软件 chrdevbaseAPP 都已经准备好了，接下来就是运行测试。为了方便测试，Linux 系统选择通过 TFTP 从网络启动，并且使用 NFS 挂载网络根文件系统，确保 uboot 中 bootcmd 环境变量的值为：

```
tftp 80800000 zImage;tftp 83000000 imx6ull-alientek-emmc.dtb;bootz 80800000 - 83000000
```

bootrags 环境变量的值为：

```
console=ttymxc0,115200 root=/dev/nfs rw nfsroot=192.168.1.250:/home/zuozhongkai/linux/nfs/rootfs ip=192.168.1.251:192.168.1.250:192.168.1.1:255.255.255.0::eth0:off
```

设置好以后启动 Linux 系统，检查开发板根文件系统中有没有“/lib/modules/4.1.15”这个目录，如果没有的话自行创建。因为是通过 NFS 将 Ubuntu 中的 rootfs(第三十八章制作好的根文件系统)目录挂载为根文件系统，所以可以很方便的将 chrdevbase.ko 和 chrdevbaseAPP 复制到 rootfs/lib/modules/4.1.15 目录中，命令如下：

```
sudo cp chrdevbase.ko chrdevbaseApp /home/zuozhongkai/linux/nfs/rootfs/lib/modules/4.1.15/ -f
```

拷贝完成以后就会在开发板的/lib/modules/4.1.15 目录下存在 chrdevbase.ko 和 chrdevbaseAPP 这两个文件，如图 40.4.4.1 所示：

```
/lib/modules/4.1.15 # ls
chrdevbase.ko  chrdevbaseApp
/lib/modules/4.1.15 #
```

图 40.4.4.1 驱动和测试文件

输入如下命令加载 chrdevbase.ko 驱动文件：

```
insmod chrdevbase.ko
```

或

```
modprobe chrdevbase.ko
```

如果使用 modprobe 加载驱动的话，可能会出现如图 40.4.4.2 所示的提示：

```
/lib/modules/4.1.15 # modprobe chrdevbase.ko
modprobe: can't open 'modules.dep': No such file or directory
/lib/modules/4.1.15 #
```

图 40.4.4.2 modprobe 错误提示

从图 40.4.4.2 可以看出，modprobe 提示无法打开“modules.dep”这个文件，因此驱动挂载失败了。我们不用手动创建 modules.dep 这个文件，直接输入 depmod 命令即可自动生成 modules.dep，有些根文件系统可能没有 depmod 这个命令，如果没有这个命令就只能重新配置 busybox，使能此命令，然后重新编译 busybox。输入“depmod”命令以后会自动生成 modules.alias、modules.symbols 和 modules.dep 这三个文件，如图 40.4.4.3 所示：

```
/lib/modules/4.1.15 # depmod
/lib/modules/4.1.15 # ls
chrdevbase.ko     modules.alias     modules.symbols
chrdevbaseApp     modules.dep
/lib/modules/4.1.15 #
```

图 40.4.4.3 depmod 命令执行结果

重新使用 modprobe 加载 chrdevbase.ko，结果如图 40.4.4.4 所示：

```
/lib/modules/4.1.15 # modprobe chrdevbase.ko
chrdevbase init!
/lib/modules/4.1.15 #
```

图 40.4.4.4 驱动加载成功

从图 40.4.4.4 可以看到"chrdevbase init!"这一行，这一行正是 chrdevbase.c 中模块入口函数 chrdevbase_init 输出的信息，说明模块加载成功！

输入"lsmod"命令即可查看当前系统中存在的模块，结果如图 40.4.4.5 所示：

```
/lib/modules/4.1.15 # lsmod
Module                  Size  Used by    Tainted: G
chrdevbase              2096  0
/lib/modules/4.1.15 #
```

图 40.4.4.5 当前系统中的模块

从图 40.4.4.5 可以看出，当前系统只有"chrdevbase"这一个模块。输入如下命令查看当前系统中有没有 chrdevbase 这个设备：

```
cat /proc/devices
```

结果如图 40.4.4.6 所示：

```
/lib/modules/4.1.15 # cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
200 chrdevbase
207 ttymxc
```

查看当前系统中的所有设备

主设备号为200的chrdevbase设备

图 40.4.4.6 当前系统设备

从图 40.4.4.6 可以看出，当前系统存在 chrdevbase 这个设备，主设备号为 200，跟我们设置的主设备号一致。

2、创建设备节点文件

驱动加载成功需要在/dev 目录下创建一个与之对应的设备节点文件，应用程序就是通过操作这个设备节点文件来完成对具体设备的操作。输入如下命令创建/dev/chrdevbase 这个设备节点文件：

```
mknod /dev/chrdevbase c 200 0
```

其中"mknod"是创建节点命令，"/dev/chrdevbase"是要创建的节点文件，"c"表示这是个

字符设备，“200”是设备的主设备号，“0”是设备的次设备号。创建完成以后就会存在/dev/chrdevbase 这个文件，可以使用“ls /dev/chrdevbase -l”命令查看，结果如图 40.4.4.7 所示：

```
/lib/modules/4.1.15 # ls /dev/chrdevbase -l
crw-r--r--    1 0        0         200,   0 Jan  1 01:25 /dev/chrdevbase
/lib/modules/4.1.15 #
```

图 40.4.4.7 /dev/chrdevbase 文件

如果 chrdevbaseAPP 想要读写 chrdevbase 设备，直接对/dev/chrdevbase 进行读写操作即可。相当于/dev/chrdevbase 这个文件是 chrdevbase 设备在用户空间中的实现。前面一直说 Linux 下一切皆文件，包括设备也是文件，现在大家应该是有这个概念了吧？

3、chrdevbase 设备操作测试

一切准备就绪，接下来就是“大考”的时刻了。使用 chrdevbaseApp 软件操作 chrdevbase 这个设备，看看读写是否正常，首先进行读操作，输入如下命令：

```
./chrdevbaseApp /dev/chrdevbase 1
```

结果如图 40.4.4.8 所示：

```
/lib/modules/4.1.15 # ./chrdevbaseApp /dev/chrdevbase 1
kernel senddata ok!
read data:kernel data!
/lib/modules/4.1.15 #
```

驱动程序中chrdevbase_read函数输出的信息
测试APP中输出的接收到的数据：kernel data!

图 40.4.4.8 读操作结果

从图 40.4.4.8 可以看出，首先输出“kernel senddata ok!”这一行信息，这是驱动程序中 chrdevbase_read 函数输出的信息，因为 chrdevbaseAPP 使用 read 函数从 chrdevbase 设备读取数据，因此 chrdevbase_read 函数就会执行。chrdevbase_read 函数向 chrdevbaseAPP 发送“kernel data!”数据，chrdevbaseAPP 接收到以后就打印出来，“read data:kernel data!”就是 chrdevbaseAPP 打印出来的接收到的数据。说明对 chrdevbase 的读操作正常，接下来测试对 chrdevbase 设备的写操作，输入如下命令：

```
./chrdevbaseApp /dev/chrdevbase 2
```

结果如图 40.4.4.9 所示：

```
/lib/modules/4.1.15 # ./chrdevbaseApp /dev/chrdevbase 2
kernel recevdata:usr data!
/lib/modules/4.1.15 #
```

图 40.4.4.9 写操作结果

只有一行“kernel recevdata:usr data!”，这个是驱动程序中的 chrdevbase_write 函数输出的。chrdevbaseAPP 使用 write 函数向 chrdevbase 设备写入数据“usr data!”。chrdevbase_write 函数接收到以后将其打印出来。说明对 chrdevbase 的写操作正常，既然读写都没问题，说明我们编写的 chrdevbase 驱动是没有问题的。

4、卸载驱动模块

如果不再使用某个设备的话可以将其驱动卸载掉，比如输入如下命令卸载掉 chrdevbase 这个设备：

```
rmmod chrdevbase.ko
```

卸载以后使用 lsmod 命令查看 chrdevbase 这个模块还存不存在，结果如图 40.4.4.10 所示：

```
/lib/modules/4.1.15 # lsmod
Module                  Size  Used by    Tainted: G
/lib/modules/4.1.15 #
```

图 40.4.4.10 系统中当前模块

从图 40.4.4.10 可以看出，此时系统已经没有任何模块了，chrdevbase 这个模块也不存在了，说明模块卸载成功。

至此，chrdevbase 这个设备的整个驱动就验证完成了，驱动工作正常。本章我们详细的讲解了字符设备驱动的开发步骤，并且以一个虚拟的 chrdevbase 设备为例，带领大家完成了第一个字符设备驱动的开发，掌握了字符设备驱动的开发框架以及测试方法，以后的字符设备驱动实验基本都以此为蓝本。

第四十一章 嵌入式 Linux LED 驱动开发实验

上一章我们详细的讲解了字符设备驱动开发步骤，并且用一个虚拟的 chrdevbase 设备为例带领大家完成了第一个字符设备驱动的开发。本章我们就开始编写第一个真正的 Linux 字符设备驱动。在 I.MX6U-ALPHA 开发板上有一个 LED 灯，我们在裸机篇中已经编写过此 LED 灯的裸机驱动，本章我们就来学习一下如何编写 Linux 下的 LED 灯驱动。

41.1 Linux 下 LED 灯驱动原理

Linux 下的任何外设驱动，最终都是要配置相应的硬件寄存器。所以本章的 LED 灯驱动最终也是对 I.MX6ULL 的 IO 口进行配置，与裸机实验不同的是，在 Linux 下编写驱动要符合 Linux 的驱动框架。I.MX6U-ALPHA 开发板上的 LED 连接到 I.MX6ULL 的 GPIO1_IO03 这个引脚上，因此本章实验的重点就是编写 Linux 下 I.MX6UL 引脚控制驱动。关于 I.MX6ULL 的 GPIO 详细讲解请参考第八章。

41.1.1 地址映射

在编写驱动之前，我们需要先简单了解一下 MMU 这个神器，MMU 全称叫做 Memory Manage Unit，也就是内存管理单元。在老版本的 Linux 中要求处理器必须有 MMU，但是现在 Linux 内核已经支持无 MMU 的处理器了。MMU 主要完成的功能如下：

①、完成虚拟空间到物理空间的映射。

②、内存保护，设置存储器的访问权限，设置虚拟存储空间的缓冲特性。

我们重点来看一下第①点，也就是虚拟空间到物理空间的映射，也叫做地址映射。首先了解两个地址概念：虚拟地址(VA,Virtual Address)、物理地址(PA，Physcical Address)。对于 32 位的处理器来说，虚拟地址范围是 2^32=4GB，我们的开发板上有 512MB 的 DDR3，这 512MB 的内存就是物理内存，经过 MMU 可以将其映射到整个 4GB 的虚拟空间，如图 41.1.1 所示：

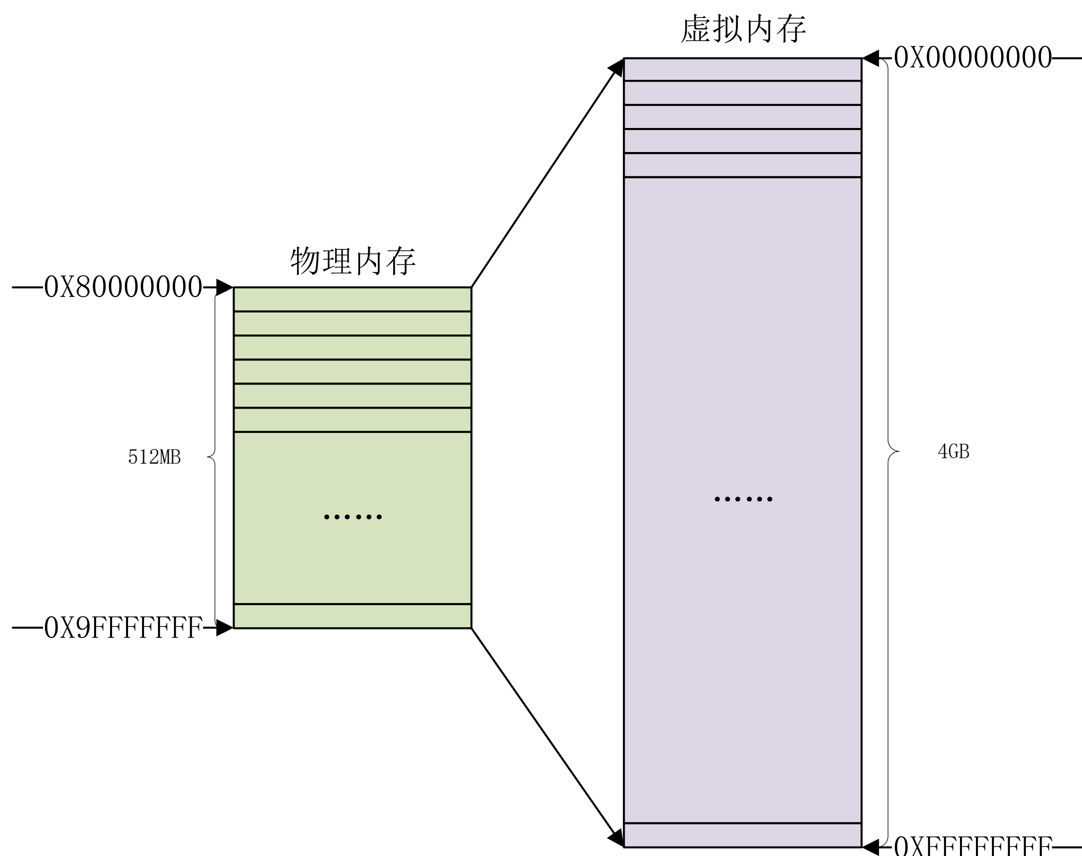


图 41.1.1 内存映射

物理内存只有 512MB，虚拟内存有 4GB，那么肯定存在多个虚拟地址映射到同一个物理地址上去，虚拟地址范围比物理地址范围大的问题处理器自会处理，这里我们不要去深究，因为 MMU 是很复杂的一个东西，后续有时间的话正点原子 Linux 团队会专门做 MMU 专题教程。

Linux 内核启动的时候会初始化 MMU，设置好内存映射，设置好以后 CPU 访问的都是虚拟地址。比如 I.MX6ULL 的 GPIO1_IO03 引脚的复用寄存器 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 的地址为 0X020E0068。如果没有开启 MMU 的话直接向 0X020E0068 这个寄存器地址写入数据就可以配置 GPIO1_IO03 的复用功能。现在开启了 MMU，并且设置了内存映射，因此就不能直接向 0X020E0068 这个地址写入数据了。我们必须得到 0X020E0068 这个物理地址在 Linux 系统里面对应的虚拟地址，这里就涉及到了物理内存和虚拟内存之间的转换，需要用到两个函数：ioremap 和 iounmap。

1、ioremap 函数

ioremap 函数用于获取指定物理地址空间对应的虚拟地址空间，定义在 arch/arm/include/asm/io.h 文件中，定义如下：

示例代码 41.1.1 ioremap 函数

```
1 #define ioremap(cookie,size) __arm_ioremap((cookie), (size),
                                              MT_DEVICE)
2
3 void __iomem * __arm_ioremap(phys_addr_t phys_addr, size_t size,
                               unsigned int mtype)
4 {
5   return arch_ioremap_caller(phys_addr, size, mtype,
                              __builtin_return_address(0));
6 }
```

ioremap 是个宏，有两个参数：cookie 和 size，真正起作用的是函数__arm_ioremap，此函数有三个参数和一个返回值，这些参数和返回值的含义如下：

phys_addr：要映射给的物理起始地址。

size：要映射的内存空间大小。

mtype：ioremap 的类型，可以选择 MT_DEVICE、MT_DEVICE_NONSHARED、MT_DEVICE_CACHED 和 MT_DEVICE_WC，ioremap 函数选择 MT_DEVICE。

返回值：__iomem 类型的指针，指向映射后的虚拟空间首地址。

假如我们要获取 I.MX6ULL 的 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 寄存器对应的虚拟地址，使用如下代码即可：

```
#define SW_MUX_GPIO1_IO03_BASE          (0X020E0068)
static void __iomem*   SW_MUX_GPIO1_IO03;
SW_MUX_GPIO1_IO03 = ioremap(GPIO1_GDIR_BASE, 4);
```

宏 SW_MUX_GPIO1_IO03_BASE 是寄存器物理地址，SW_MUX_GPIO1_IO03 是映射后的虚拟地址。对于 I.MX6ULL 来说一个寄存器是 4 字节(32 位)的，因此映射的内存长度为 4。映射完成以后直接对 SW_MUX_GPIO1_IO03 进行读写操作即可。

2、iounmap 函数

卸载驱动的时候需要使用 iounmap 函数释放掉 ioremap 函数所做的映射，iounmap 函数原型如下：

示例代码 41.1.2 iounmap 函数原型

```
void iounmap (volatile void __iomem *addr)
```

iounmap 只有一个参数 addr，此参数就是要取消映射的虚拟地址空间首地址。假如我们现在要取消掉 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 寄存器的地址映射，使用如下代码即可：

```
iounmap(SW_MUX_GPIO1_IO03);
```

41.1.2 I/O 内存访问函数

这里说的 I/O 是输入/输出的意思，并不是我们学习单片机的时候讲的 GPIO 引脚。这里涉及到两个概念：I/O 端口和 I/O 内存。当外部寄存器或内存映射到 IO 空间时，称为 I/O 端口。当外部寄存器或内存映射到内存空间时，称为 I/O 内存。但是对于 ARM 来说没有 I/O 空间这个概念，因此 ARM 体系下只有 I/O 内存(可以直接理解为内存)。使用 ioremap 函数将寄存器的物理地址映射到虚拟地址以后，我们就可以直接通过指针访问这些地址，但是 Linux 内核不建议这么做，而是推荐使用一组操作函数来对映射后的内存进行读写操作。

1、读操作函数

读操作函数有如下几个：

示例代码 41.1.2.1 读操作函数

```
1 u8  readb(const volatile void __iomem *addr)
2 u16 readw(const volatile void __iomem *addr)
3 u32 readl(const volatile void __iomem *addr)
```

readb、readw 和 readl 这三个函数分别对应 8bit、16bit 和 32bit 读操作，参数 addr 就是要读取写内存地址，返回值就是读取到的数据。

2、写操作函数

写操作函数有如下几个：

示例代码 41.1.2.2 写操作函数

```
1 void writeb(u8 value,  volatile void __iomem *addr)
2 void writew(u16 value, volatile void __iomem *addr)
3 void writel(u32 value, volatile void __iomem *addr)
```

writeb、writew 和 writel 这三个函数分别对应 8bit、16bit 和 32bit 写操作，参数 value 是要写入的数值，addr 是要写入的地址。

41.2 硬件原理图分析

本章实验硬件原理图参考 8.3 小节即可。

41.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 2_led。

本章实验编写 Linux 下的 LED 灯驱动，可以通过应用程序对 I.MX6U-ALPHA 开发板上的 LED 灯进行开关操作。

41.3.1 LED 灯驱动程序编写

新建名为“2_led”文件夹，然后在 2_led 文件夹里面创建 VSCode 工程，工作区命名为“led”。工程创建好以后新建 led.c 文件，此文件就是 led 的驱动文件，在 led.c 里面输入如下内容：

示例代码 41.3.1.1 led.c 驱动文件代码

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <asm/mach/map.h>
10  #include <asm/uaccess.h>
11  #include <asm/io.h>
12  /***************************************************************
13  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
14  文件名      : led.c
15  作者        : 左忠凯
16  版本        : V1.0
17  描述        : LED 驱动文件。
18  其他        : 无
19  论坛        : www.openedv.com
20  日志        : 初版 V1.0 2019/1/30 左忠凯创建
21  ***************************************************************/
22  #define LED_MAJOR     200         /* 主设备号 */
23  #define LED_NAME      "led"       /* 设备名字 */
24
25  #define LEDOFF        0           /* 关灯     */
26  #define LEDON         1           /* 开灯     */
27
28  /* 寄存器物理地址 */
29  #define CCM_CCGR1_BASE              (0X020C406C)
30  #define SW_MUX_GPIO1_IO03_BASE      (0X020E0068)
31  #define SW_PAD_GPIO1_IO03_BASE      (0X020E02F4)
32  #define GPIO1_DR_BASE               (0X0209C000)
33  #define GPIO1_GDIR_BASE             (0X0209C004)
34
35  /* 映射后的寄存器虚拟地址指针 */
36  static void __iomem *IMX6U_CCM_CCGR1;
37  static void __iomem *SW_MUX_GPIO1_IO03;
38  static void __iomem *SW_PAD_GPIO1_IO03;
39  static void __iomem *GPIO1_DR;
40  static void __iomem *GPIO1_GDIR;
41
42  /*
43   * @description   : LED 打开/关闭
```

```
44   * @param - sta   : LEDON(0) 打开 LED，LEDOFF(1) 关闭 LED
45   * @return        : 无
46   */
47  void led_switch(u8 sta)
48  {
49      u32 val = 0;
50      if(sta == LEDON) {
51          val = readl(GPIO1_DR);
52          val &= ~(1 << 3);
53          writel(val, GPIO1_DR);
54      }else if(sta == LEDOFF) {
55          val = readl(GPIO1_DR);
56          val|= (1 << 3);
57          writel(val, GPIO1_DR);
58      }
59  }
60
61  /*
62   * @description   : 打开设备
63   * @param – inode : 传递给驱动的 inode
64   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
65   *                 一般在 open 的时候将 private_data 指向设备结构体。
66   * @return        : 0 成功;其他 失败
67   */
68  static int led_open(struct inode *inode, struct file *filp)
69  {
70      return 0;
71  }
72
73  /*
74   * @description   : 从设备读取数据
75   * @param - filp  : 要打开的设备文件(文件描述符)
76   * @param - buf   : 返回给用户空间的数据缓冲区
77   * @param - cnt   : 要读取的数据长度
78   * @param - offt  : 相对于文件首地址的偏移
79   * @return        : 读取的字节数，如果为负值，表示读取失败
80   */
81  static ssize_t led_read(struct file *filp, char __user *buf,
                         size_t cnt, loff_t *offt)
82  {
83      return 0;
84  }
85
```

```
86 /*
87  * @description  : 向设备写数据
88  * @param - filp : 设备文件，表示打开的文件描述符
89  * @param - buf  : 要写给设备写入的数据
90  * @param - cnt  : 要写入的数据长度
91  * @param - offt : 相对于文件首地址的偏移
92  * @return       : 写入的字节数，如果为负值，表示写入失败
93  */
94 static ssize_t led_write(struct file *filp, const char __user *buf,
                         size_t cnt, loff_t *offt)
95 {
96     int retvalue;
97     unsigned char databuf[1];
98     unsigned char ledstat;
99 
100    retvalue = copy_from_user(databuf, buf, cnt);
101    if(retvalue < 0) {
102        printk("kernel write failed!\r\n");
103        return -EFAULT;
104    }
105 
106    ledstat = databuf[0];       /* 获取状态值   */
107 
108    if(ledstat == LEDON) {
109        led_switch(LEDON);      /* 打开 LED 灯  */
110    } else if(ledstat == LEDOFF) {
111        led_switch(LEDOFF);     /* 关闭 LED 灯  */
112    }
113    return 0;
114 }
115 
116 /*
117  * @description  : 关闭/释放设备
118  * @param - filp : 要关闭的设备文件(文件描述符)
119  * @return       : 0 成功;其他 失败
120  */
121 static int led_release(struct inode *inode, struct file *filp)
122 {
123    return 0;
124 }
125 
126 /* 设备操作函数 */
127 static struct file_operations led_fops = {
```

```
128     .owner = THIS_MODULE,
129     .open = led_open,
130     .read = led_read,
131     .write = led_write,
132     .release =  led_release,
133 };
134 
135 /*
136  * @description   : 驱动出口函数
137  * @param         : 无
138  * @return        : 无
139  */
140 static int __init led_init(void)
141 {
142     int retvalue = 0;
143     u32 val = 0;
144 
145     /* 初始化 LED */
146     /* 1、寄存器地址映射 */
147     IMX6U_CCM_CCGR1 = ioremap(CCM_CCGR1_BASE, 4);
148     SW_MUX_GPIO1_IO03 = ioremap(SW_MUX_GPIO1_IO03_BASE, 4);
149     SW_PAD_GPIO1_IO03 = ioremap(SW_PAD_GPIO1_IO03_BASE, 4);
150     GPIO1_DR = ioremap(GPIO1_DR_BASE, 4);
151     GPIO1_GDIR = ioremap(GPIO1_GDIR_BASE, 4);
152 
153     /* 2、使能 GPIO1 时钟 */
154     val = readl(IMX6U_CCM_CCGR1);
155     val &= ~(3 << 26);  /* 清除以前的设置 */
156     val |= (3 << 26);   /* 设置新值 */
157     writel(val, IMX6U_CCM_CCGR1);
158 
159     /* 3、设置 GPIO1_IO03 的复用功能，将其复用为
160      *    GPIO1_IO03，最后设置 IO 属性。
161      */
162     writel(5, SW_MUX_GPIO1_IO03);
163 
164     /* 寄存器 SW_PAD_GPIO1_IO03 设置 IO 属性 */
165     writel(0x10B0, SW_PAD_GPIO1_IO03);
166 
167     /* 4、设置 GPIO1_IO03 为输出功能 */
168     val = readl(GPIO1_GDIR);
169     val &= ~(1 << 3);   /* 清除以前的设置 */
170     val |= (1 << 3);    /* 设置为输出 */
```

```
171     writel(val, GPIO1_GDIR);
172 
173     /* 5、默认关闭 LED */
174     val = readl(GPIO1_DR);
175     val |= (1 << 3);
176     writel(val, GPIO1_DR);
177 
178     /* 6、注册字符设备驱动 */
179     retvalue = register_chrdev(LED_MAJOR, LED_NAME, &led_fops);
180     if(retvalue < 0){
181         printk("register chrdev failed!\r\n");
182         return -EIO;
183     }
184     return 0;
185 }
186 
187 /*
188  * @description   : 驱动出口函数
189  * @param         : 无
190  * @return        : 无
191  */
192 static void __exit led_exit(void)
193 {
194     /* 取消映射 */
195     iounmap(IMX6U_CCM_CCGR1);
196     iounmap(SW_MUX_GPIO1_IO03);
197     iounmap(SW_PAD_GPIO1_IO03);
198     iounmap(GPIO1_DR);
199     iounmap(GPIO1_GDIR);
200 
201     /* 注销字符设备驱动 */
202     unregister_chrdev(LED_MAJOR, LED_NAME);
203 }
204 
205 module_init(led_init);
206 module_exit(led_exit);
207 MODULE_LICENSE("GPL");
208 MODULE_AUTHOR("zuozhongkai");
```

第 22~26 行，定义了一些宏，包括主设备号、设备名字、LED 开/关宏。

第 29~33 行，本实验要用到的寄存器宏定义。

第 36~40 行，经过内存映射以后的寄存器地址指针。

第 47~59 行，led_switch 函数，用于控制开发板上的 LED 灯亮灭，当参数 sta 为 LEDON(0)的时候打开 LED 灯，sta 为 LEDOFF(1)的时候关闭 LED 灯。

第 68~71 行，led_open 函数，为空函数，可以自行在此函数中添加相关内容，一般在此函数中将设备结构体作为参数 filp 的私有数据(filp->private_data)。

第 81~84 行，led_read 函数，为空函数，如果想在应用程序中读取 LED 的状态，那么就可以在此函数中添加相应的代码，比如读取 GPIO1_DR 寄存器的值，然后返回给应用程序。

第 94~114 行，led_write 函数，实现对 LED 灯的开关操作，当应用程序调用 write 函数向 led 设备写数据的时候此函数就会执行。首先通过函数 copy_from_user 获取应用程序发送过来的操作信息(打开还是关闭 LED)，最后根据应用程序的操作信息来打开或关闭 LED 灯。

第 121~124 行，led_release 函数，为空函数，可以自行在此函数中添加相关内容，一般关闭设备的时候会释放掉 led_open 函数中添加的私有数据。

第 127~133 行，设备文件操作结构体 led_fops 的定义和初始化。

第 140~185 行，驱动入口函数 led_init，此函数实现了 LED 的初始化工作，147~151 行通过 ioremap 函数获取物理寄存器地址映射后的虚拟地址，得到寄存器对应的虚拟地址以后就可以完成相关初始化工作了。比如是能 GPIO1 时钟、设置 GPIO1_IO03 复用功能、配置 GPIO1_IO03 的属性等等。最后，最重要的一步！使用 register_chrdev 函数注册 led 这个字符设备。

第 192~202 行，驱动出口函数 led_exit，首先使用函数 iounmap 取消内存映射，最后使用函数 unregister_chrdev 注销 led 这个字符设备。

第 205~206 行，使用 module_init 和 module_exit 这两个函数指定 led 设备驱动加载和卸载函数。

第 207~208 行，添加 LICENSE 和作者信息。

41.3.2 编写测试 APP

编写测试 APP，led 驱动加载成功以后手动创建/dev/led 节点，应用 APP 通过操作/dev/led 文件来完成对 LED 设备的控制。向/dev/led 文件写 0 表示关闭 LED 灯，写 1 表示打开 LED 灯。新建 ledApp.c 文件，在里面输入如下内容：

示例代码 41.3.2.1 ledApp.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名      : ledApp.c
11 作者        : 左忠凯
12 版本        : V1.0
13 描述        : LED 驱动测试 APP。
14 其他        : 无
15 使用方法    : ./ledtest /dev/led  0 关闭 LED
16               ./ledtest /dev/led  1 打开 LED
17 论坛        : www.openedv.com
18 日志        : 初版 V1.0 2019/1/30 左忠凯创建
```

```
19 *****************************************************************/
20 
21 #define LEDOFF   0
22 #define LEDON    1
23 
24 /*
25  * @description    : main 主程序
26  * @param - argc   : argv 数组元素个数
27  * @param - argv   : 具体参数
28  * @return         : 0 成功;其他 失败
29  */
30 int main(int argc, char *argv[])
31 {
32     int fd, retvalue;
33     char *filename;
34     unsigned char databuf[1];
35 
36     if(argc != 3){
37         printf("Error Usage!\r\n");
38         return -1;
39     }
40 
41     filename = argv[1];
42 
43     /* 打开 led 驱动 */
44     fd = open(filename, O_RDWR);
45     if(fd < 0){
46         printf("file %s open failed!\r\n", argv[1]);
47         return -1;
48     }
49 
50     databuf[0] = atoi(argv[2]); /* 要执行的操作：打开或关闭 */
51 
52     /* 向/dev/led 文件写入数据 */
53     retvalue = write(fd, databuf, sizeof(databuf));
54     if(retvalue < 0){
55         printf("LED Control Failed!\r\n");
56         close(fd);
57         return -1;
58     }
59 
60     retvalue = close(fd); /* 关闭文件 */
61     if(retvalue < 0){
```

```
62          printf("file %s close failed!\r\n", argv[1]);
63          return -1;
64      }
65      return 0;
66 }
```

ledApp.c 的内容还是很简单的，就是对 led 的驱动文件进行最基本的打开、关闭、写操作等。

41.4 运行测试

41.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 led.o，Makefile 内容如下所示：

示例代码 41.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := led.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 led.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“led.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

41.4.2 运行测试

将上一小节编译出来的 led.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 led.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe led.ko         //加载驱动
```

驱动加载成功以后创建“/dev/led”设备节点，命令如下：

```
mknod /dev/led c 200 0
```

驱动节点创建成功以后就可以使用 ledApp 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/led 1        //打开 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否点亮，如果点亮的话

说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp /dev/led 0        //关闭 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否熄灭，如果熄灭的话说明我们编写的 LED 驱动工作完全正常！至此，我们成功编写了第一个真正的 Linux 驱动设备程序。

如果要卸载驱动的话输入如下命令即可：

```
rmmod led.ko
```

第四十二章 新字符设备驱动实验

经过前两章实验的实战操作，我们已经掌握了 Linux 字符设备驱动开发的基本步骤，字符设备驱动开发重点是使用 register_chrdev 函数注册字符设备，当不再使用设备的时候就使用 unregister_chrdev 函数注销字符设备，驱动模块加载成功以后还需要手动使用 mknod 命令创建设备节点。register_chrdev 和 unregister_chrdev 这两个函数是老版本驱动使用的函数，现在新的字符设备驱动已经不再使用这两个函数，而是使用 Linux 内核推荐的新字符设备驱动 API 函数。本节我们就来学习一下如何编写新字符设备驱动，并且在驱动模块加载的时候自动创建设备节点文件。

42.1 新字符设备驱动原理

42.1.1 分配和释放设备号

使用 register_chrdev 函数注册字符设备的时候只需要给定一个主设备号即可，但是这样会带来两个问题：

①、需要我们事先确定好哪些主设备号没有使用。

②、会将一个主设备号下的所有次设备号都使用掉，比如现在设置 LED 这个主设备号为 200，那么 0~1048575(2^20-1)这个区间的次设备号就全部都被 LED 一个设备分走了。这样太浪费次设备号了！一个 LED 设备肯定只能有一个主设备号，一个次设备号。

解决这两个问题最好的方法就是要使用设备号的时候向 Linux 内核申请，需要几个就申请几个，由 Linux 内核分配设备可以使用的设备号。这个就是我们在 40.3.2 小节讲解的设备号的分配，如果没有指定设备号的话就使用如下函数来申请设备号：

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

如果给定了设备的主设备号和次设备号就使用如下所示函数来注册设备号即可：

```
int register_chrdev_region(dev_t from, unsigned count, const char *name)
```

参数 from 是要申请的起始设备号，也就是给定的设备号；参数 count 是要申请的数量，一般都是一个；参数 name 是设备名字。

注销字符设备之后要释放掉设备号，不管是通过 alloc_chrdev_region 函数还是 register_chrdev_region 函数申请的设备号，统一使用如下释放函数：

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

新字符设备驱动下，设备号分配示例代码如下：

示例代码 42.1.1.1 新字符设备驱动下设备号分配

```
1  int major;                          /* 主设备号              */
2  int minor;                          /* 次设备号              */
3  dev_t devid;                        /* 设备号                */
4
5  if (major) {                        /*  定义了主设备号       */
6      devid = MKDEV(major, 0);        /* 大部分驱动次设备号都选择 0 */
7      register_chrdev_region(devid, 1, "test");
8  } else {                            /* 没有定义设备号        */
9      alloc_chrdev_region(&devid, 0, 1, "test");  /* 申请设备号 */
10     major = MAJOR(devid);           /* 获取分配号的主设备号  */
11     minor = MINOR(devid);           /* 获取分配号的次设备号  */
12 }
```

第 1~3 行，定义了主/次设备号变量 major 和 minor，以及设备号变量 devid。

第 5 行，判断主设备号 major 是否有效，在 Linux 驱动中一般给出主设备号的话就表示这个设备的设备号已经确定了，因为次设备号基本上都选择 0，这算个 Linux 驱动开发中约定俗成的一种规定了。

第 6 行，如果 major 有效的话就使用 MKDEV 来构建设备号，次设备号选择 0。

第 7 行，使用 register_chrdev_region 函数来注册设备号。

第 9~11 行，如果 major 无效，那就表示没有给定设备号。此时就要使用 alloc_chrdev_region 函数来申请设备号。设备号申请成功以后使用 MAJOR 和 MINOR 来提取出主设备号和次设备

号，当然了，第 10 和 11 行提取主设备号和次设备号的代码可以不要。

如果要注销设备号的话，使用如下代码即可：

示例代码 42.1.1.2 cdev 结构体

```
1 unregister_chrdev_region(devid, 1);          /* 注销设备号 */
```

注销设备号的代码很简单。

42.1.2 新的字符设备注册方法

1、字符设备结构

在 Linux 中使用 cdev 结构体表示一个字符设备，cdev 结构体在 include/linux/cdev.h 文件中的定义如下：

示例代码 42.1.2.1 cdev 结构体

```
1 struct cdev {
2      struct kobject    kobj;
3      struct module     *owner;
4      const struct file_operations *ops;
5      struct list_head  list;
6      dev_t             dev;
7      unsigned int      count;
8 };
```

在 cdev 中有两个重要的成员变量：ops 和 dev，这两个就是字符设备文件操作函数集合 file_operations 以及设备号 dev_t。编写字符设备驱动之前需要定义一个 cdev 结构体变量，这个变量就表示一个字符设备，如下所示：

```
struct cdev test_cdev;
```

2、cdev_init 函数

定义好 cdev 变量以后就要使用 cdev_init 函数对其进行初始化，cdev_init 函数原型如下：

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

参数 cdev 就是要初始化的 cdev 结构体变量，参数 fops 就是字符设备文件操作函数集合。使用 cdev_init 函数初始化 cdev 变量的示例代码如下：

示例代码 42.1.2.2 cdev_init 函数使用示例代码

```
1  struct cdev testcdev;
2 
3  /* 设备操作函数 */
4  static struct file_operations test_fops = {
5      .owner = THIS_MODULE,
6      /* 其他具体的初始项 */
7  };
8 
9  testcdev.owner = THIS_MODULE;
10 cdev_init(&testcdev, &test_fops); /* 初始化 cdev 结构体变量 */
```

3、cdev_add 函数

cdev_add 函数用于向 Linux 系统添加字符设备(cdev 结构体变量)，首先使用 cdev_init 函数

完成对 cdev 结构体变量的初始化，然后使用 cdev_add 函数向 Linux 系统添加这个字符设备。cdev_add 函数原型如下：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count)
```

参数 p 指向要添加的字符设备(cdev 结构体变量)，参数 dev 就是设备所使用的设备号，参数 count 是要添加的设备数量。完善示例代码 42.1.2.2，加入 cdev_add 函数，内容如下所示：

示例代码 42.1.2.2 cdev_add 函数使用示例

```
1  struct cdev testcdev;
2  
3  /* 设备操作函数 */
4  static struct file_operations test_fops = {
5       .owner = THIS_MODULE,
6       /* 其他具体的初始项 */
7  };
8  
9  testcdev.owner = THIS_MODULE;
10 cdev_init(&testcdev, &test_fops);       /* 初始化 cdev 结构体变量    */
11 cdev_add(&testcdev, devid, 1);          /* 添加字符设备              */
```

示例代码 42.1.2.2 就是新的注册字符设备代码段，Linux 内核中大量的字符设备驱动都是采用这种方法向 Linux 内核添加字符设备。如果在加上示例代码 42.1.1.1 中分配设备号的程序，那么就它们一起实现的就是函数 register_chrdev 的功能。

3、cdev_del 函数

卸载驱动的时候一定要使用 cdev_del 函数从 Linux 内核中删除相应的字符设备，cdev_del 函数原型如下：

```
void cdev_del(struct cdev *p)
```

参数 p 就是要删除的字符设备。如果要删除字符设备，参考如下代码：

示例代码 42.1.2.3 cdev_del 函数使用示例

```
1 cdev_del(&testcdev);   /*  删除 cdev */
```

cdev_del 和 unregister_chrdev_region 这两个函数合起来的功能相当于 unregister_chrdev 函数。

42.2 自动创建设备节点

在前面的 Linux 驱动实验中，当我们使用 modprobe 加载驱动程序以后还需要使用命令“mknod”手动创建设备节点。本节就来讲解一下如何实现自动创建设备节点，在驱动中实现自动创建设备节点的功能以后，使用 modprobe 加载驱动模块成功的话就会自动在/dev 目录下创建对应的设备文件。

42.2.1 mdev 机制

udev 是一个用户程序，在 Linux 下通过 udev 来实现设备文件的创建与删除，udev 可以检测系统中硬件设备状态，可以根据系统中硬件设备状态来创建或者删除设备文件。比如使用 modprobe 命令成功加载驱动模块以后就自动在/dev 目录下创建对应的设备节点文件,使用 rmmod 命令卸载驱动模块以后就删除掉/dev 目录下的设备节点文件。使用 busybox 构建根文件系统的时候，busybox 会创建一个 udev 的简化版本—mdev，所以在嵌入式 Linux 中我们使用

mdev 来实现设备节点文件的自动创建与删除，Linux 系统中的热插拔事件也由 mdev 管理，在/etc/init.d/rcS 文件中如下语句：

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

上述命令设置热插拔事件由 mdev 来管理，关于 udev 或 mdev 更加详细的工作原理这里就不详细探讨了，我们重点来学习一下如何通过 mdev 来实现设备文件节点的自动创建与删除。

42.2.1 创建和删除类

自动创建设备节点的工作是在驱动程序的入口函数中完成的，一般在 cdev_add 函数后面添加自动创建设备节点相关代码。首先要创建一个 class 类，class 是个结构体，定义在文件 include/linux/device.h 里面。class_create 是类创建函数，class_create 是个宏定义，内容如下：

示例代码 42.2.1.1 class_create 函数

```
1 #define class_create(owner, name)     \
2 ({                                   \
3   static struct lock_class_key __key; \
4   __class_create(owner, name, &__key);   \
5 })
6
7 struct class *__class_create(struct module *owner, const char *name,
8                               struct lock_class_key *key)
```

根据上述代码，将宏 class_create 展开以后内容如下：

```
struct class *class_create (struct module *owner, const char *name)
```

class_create 一共有两个参数，参数 owner 一般为 THIS_MODULE，参数 name 是类名字。返回值是个指向结构体 class 的指针，也就是创建的类。

卸载驱动程序的时候需要删除掉类，类删除函数为 class_destroy，函数原型如下：

```
void class_destroy(struct class *cls);
```

参数 cls 就是要删除的类。

42.2.2 创建设备

上一小节创建好类以后还不能实现自动创建设备节点，我们还需要在这个类下创建一个设备。使用 device_create 函数在类下面创建设备，device_create 函数原型如下：

```
struct device *device_create(struct class     *class,
                             struct device    *parent,
                             dev_t            devt,
                             void             *drvdata,
                             const char       *fmt, ...)
```

device_create 是个可变参数函数，参数 class 就是设备要创建哪个类下面；参数 parent 是父设备，一般为 NULL，也就是没有父设备；参数 devt 是设备号；参数 drvdata 是设备可能会使用的一些数据，一般为 NULL；参数 fmt 是设备名字，如果设置 fmt=xxx 的话，就会生成/dev/xxx 这个设备文件。返回值就是创建好的设备。

同样的，卸载驱动的时候需要删除掉创建的设备，设备删除函数为 device_destroy，函数原型如下：

```
void device_destroy(struct class *class, dev_t devt)
```

参数 classs 是要删除的设备所处的类，参数 devt 是要删除的设备号。

42.2.3 参考示例

在驱动入口函数里面创建类和设备，在驱动出口函数里面删除类和设备，参考示例如下：

示例代码 42.2.3.1 创建/删除类/设备参考代码

```
1  struct class *class;      /* 类       */
2  struct device *device;    /* 设备     */
3  dev_t devid;              /* 设备号   */
4
5  /* 驱动入口函数 */
6  static int __init xxx_init(void)
7  {
8      /* 创建类   */
9      class = class_create(THIS_MODULE, "xxx");
10     /* 创建设备 */
11     device = device_create(class, NULL, devid, NULL, "xxx");
12     return 0;
13 }
14
15 /* 驱动出口函数 */
16 static void __exit led_exit(void)
17 {
18     /* 删除设备 */
19     device_destroy(newchrled.class, newchrled.devid);
20     /* 删除类   */
21     class_destroy(newchrled.class);
22 }
23
24 module_init(led_init);
25 module_exit(led_exit);
```

42.3 设置文件私有数据

每个硬件设备都有一些属性，比如主设备号(dev_t)，类(class)、设备(device)、开关状态(state)等等，在编写驱动的时候你可以将这些属性全部写成变量的形式，如下所示：

示例代码 42.3.1 变量形式的设备属性

```
dev_t devid;              /* 设备号   */
struct cdev cdev;         /* cdev     */
struct class *class;      /* 类       */
struct device *device;    /* 设备     */
int major;                /* 主设备号 */
int minor;                /* 次设备号 */
```

这样写肯定没有问题，但是这样写不专业！对于一个设备的所有属性信息我们最好将其做

成一个结构体。编写驱动 open 函数的时候将设备结构体作为私有数据添加到设备文件中，如下所示：

示例代码 42.3.2 设备结构体作为私有数据

```
  /* 设备结构体 */
1  struct test_dev{
2       dev_t devid;            /* 设备号      */
3       struct cdev cdev;       /* cdev       */
4       struct class *class;    /* 类         */
5       struct device *device;  /* 设备        */
6       int major;              /* 主设备号    */
7       int minor;              /* 次设备号    */
8  };
9
10 struct test_dev testdev;
11
12 /* open 函数 */
13 static int test_open(struct inode *inode, struct file *filp)
14 {
15      filp->private_data = &testdev; /* 设置私有数据 */
16      return 0;
17 }
```

在 open 函数里面设置好私有数据以后，在 write、read、close 等函数中直接读取 private_data 即可得到设备结构体。

42.4 硬件原理图分析

本章实验硬件原理图参考 8.3 小节即可。

42.5 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 3_newchrled。

本章实验在上一章实验的基础上完成，重点是使用了新的字符设备驱动、设置了文件私有数据、添加了自动创建设备节点相关内容。

42.5.1 LED 灯驱动程序编写

新建名为“3_newchrled”文件夹，然后在 3_newchrled 文件夹里面创建 vscode 工程，工作区命名为“newchrled”。工程创建好以后新建 newchrled.c 文件，在 newchrled.c 里面输入如下内容：

示例代码 42.5.1.1 newchrled.c 文件

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
```

```
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <asm/mach/map.h>
12  #include <asm/uaccess.h>
13  #include <asm/io.h>
14
15  /***************************************************************
16  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
17  文件名      : newchrled.c
18  作者        : 左忠凯
19  版本        : V1.0
20  描述        : LED 驱动文件。
21  其他        : 无
22  论坛        : www.openedv.com
23  日志        : 初版 V1.0 2019/6/27 左忠凯创建
24  ***************************************************************/
25  #define NEWCHRLED_CNT        1             /* 设备号个数   */
26  #define NEWCHRLED_NAME       "newchrled"   /* 名字         */
27  #define LEDOFF               0             /* 关灯         */
28  #define LEDON                1             /* 开灯         */
29
30  /* 寄存器物理地址 */
31  #define CCM_CCGR1_BASE               (0X020C406C)
32  #define SW_MUX_GPIO1_IO03_BASE       (0X020E0068)
33  #define SW_PAD_GPIO1_IO03_BASE       (0X020E02F4)
34  #define GPIO1_DR_BASE                (0X0209C000)
35  #define GPIO1_GDIR_BASE              (0X0209C004)
36
37  /* 映射后的寄存器虚拟地址指针 */
38  static void __iomem *IMX6U_CCM_CCGR1;
39  static void __iomem *SW_MUX_GPIO1_IO03;
40  static void __iomem *SW_PAD_GPIO1_IO03;
41  static void __iomem *GPIO1_DR;
42  static void __iomem *GPIO1_GDIR;
43
44  /* newchrled 设备结构体 */
45  struct newchrled_dev{
46      dev_t devid;              /* 设备号   */
47      struct cdev cdev;         /* cdev     */
48      struct class *class;      /* 类       */
```

```
49      struct device *device;      /* 设备     */
50      int major;                  /* 主设备号 */
51      int minor;                  /* 次设备号 */
52  };
53
54  struct newchrled_dev newchrled; /* led 设备 */
55
56  /*
57   * @description   : LED 打开/关闭
58   * @param - sta   : LEDON(0) 打开 LED，LEDOFF(1) 关闭 LED
59   * @return        : 无
60   */
61  void led_switch(u8 sta)
62  {
63      u32 val = 0;
64      if(sta == LEDON) {
65          val = readl(GPIO1_DR);
66          val &= ~(1 << 3);
67          writel(val, GPIO1_DR);
68      }else if(sta == LEDOFF) {
69          val = readl(GPIO1_DR);
70          val|= (1 << 3);
71          writel(val, GPIO1_DR);
72      }
73  }
74
75  /*
76   * @description   : 打开设备
77   * @param – inode : 传递给驱动的 inode
78   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
79   *                 一般在 open 的时候将 private_data 指向设备结构体。
80   * @return        : 0 成功;其他 失败
81   */
82  static int led_open(struct inode *inode, struct file *filp)
83  {
84      filp->private_data = &newchrled; /* 设置私有数据 */
85      return 0;
86  }
87
88  /*
89   * @description   : 从设备读取数据
90   * @param - filp  : 要打开的设备文件(文件描述符)
91   * @param - buf   : 返回给用户空间的数据缓冲区
```

```
92   * @param - cnt   : 要读取的数据长度
93   * @param - offt  : 相对于文件首地址的偏移
94   * @return        : 读取的字节数，如果为负值，表示读取失败
95   */
96  static ssize_t led_read(struct file *filp, char __user *buf,
                             size_t cnt, loff_t *offt)
97  {
98      return 0;
99  }
100 
101 /*
102  * @description   : 向设备写数据
103  * @param - filp  : 设备文件，表示打开的文件描述符
104  * @param - buf   : 要写给设备写入的数据
105  * @param - cnt   : 要写入的数据长度
106  * @param - offt  : 相对于文件首地址的偏移
107  * @return        : 写入的字节数，如果为负值，表示写入失败
108  */
109 static ssize_t led_write(struct file *filp, const char __user *buf,
                             size_t cnt, loff_t *offt)
110 {
111     int retvalue;
112     unsigned char databuf[1];
113     unsigned char ledstat;
114 
115     retvalue = copy_from_user(databuf, buf, cnt);
116     if(retvalue < 0) {
117         printk("kernel write failed!\r\n");
118         return -EFAULT;
119     }
120 
121     ledstat = databuf[0];       /* 获取状态值   */
122 
123     if(ledstat == LEDON) {
124         led_switch(LEDON);      /* 打开 LED 灯  */
125     } else if(ledstat == LEDOFF) {
126         led_switch(LEDOFF);     /* 关闭 LED 灯  */
127     }
128     return 0;
129 }
130 
131 /*
132  * @description   : 关闭/释放设备
```

```
133  * @param - filp : 要关闭的设备文件(文件描述符)
134  * @return       : 0 成功;其他 失败
135  */
136 static int led_release(struct inode *inode, struct file *filp)
137 {
138     return 0;
139 }
140 
141 /* 设备操作函数 */
142 static struct file_operations newchrled_fops = {
143     .owner = THIS_MODULE,
144     .open = led_open,
145     .read = led_read,
146     .write = led_write,
147     .release =  led_release,
148 };
149 
150 /*
151  * @description   : 驱动出口函数
152  * @param         : 无
153  * @return        : 无
154  */
155 static int __init led_init(void)
156 {
157     u32 val = 0;
158 
159     /* 初始化 LED */
160     /* 1、寄存器地址映射 */
161     IMX6U_CCM_CCGR1 = ioremap(CCM_CCGR1_BASE, 4);
162     SW_MUX_GPIO1_IO03 = ioremap(SW_MUX_GPIO1_IO03_BASE, 4);
163     SW_PAD_GPIO1_IO03 = ioremap(SW_PAD_GPIO1_IO03_BASE, 4);
164     GPIO1_DR = ioremap(GPIO1_DR_BASE, 4);
165     GPIO1_GDIR = ioremap(GPIO1_GDIR_BASE, 4);
166 
167     /* 2、使能 GPIO1 时钟 */
168     val = readl(IMX6U_CCM_CCGR1);
169     val &= ~(3 << 26);  /* 清楚以前的设置 */
170     val |= (3 << 26);   /* 设置新值 */
171     writel(val, IMX6U_CCM_CCGR1);
172 
173     /* 3、设置 GPIO1_IO03 的复用功能，将其复用为
174      *    GPIO1_IO03，最后设置 IO 属性。
175      */
```

```
176     writel(5, SW_MUX_GPIO1_IO03);
177 
178     /* 寄存器 SW_PAD_GPIO1_IO03 设置 IO 属性 */
179     writel(0x10B0, SW_PAD_GPIO1_IO03);
180 
181     /* 4、设置 GPIO1_IO03 为输出功能 */
182     val = readl(GPIO1_GDIR);
183     val &= ~(1 << 3);  /* 清除以前的设置 */
184     val |= (1 << 3);   /* 设置为输出 */
185     writel(val, GPIO1_GDIR);
186 
187     /* 5、默认关闭 LED */
188     val = readl(GPIO1_DR);
189     val |= (1 << 3);
190     writel(val, GPIO1_DR);
191 
192     /* 注册字符设备驱动 */
193     /* 1、创建设备号 */
194     if (newchrled.major) {      /*  定义了设备号 */
195         newchrled.devid = MKDEV(newchrled.major, 0);
196         register_chrdev_region(newchrled.devid, NEWCHRLED_CNT,
                                   NEWCHRLED_NAME);
197     } else {                    /* 没有定义设备号 */
198         alloc_chrdev_region(&newchrled.devid, 0, NEWCHRLED_CNT,
                                NEWCHRLED_NAME);      /* 申请设备号 */
199         newchrled.major = MAJOR(newchrled.devid); /* 获取主设备号 */
200         newchrled.minor = MINOR(newchrled.devid); /* 获取次设备号 */
201     }
202     printk("newcheled major=%d,minor=%d\r\n",newchrled.major,
                newchrled.minor);
203 
204     /* 2、初始化 cdev */
205     newchrled.cdev.owner = THIS_MODULE;
206     cdev_init(&newchrled.cdev, &newchrled_fops);
207 
208     /* 3、添加一个 cdev */
209     cdev_add(&newchrled.cdev, newchrled.devid, NEWCHRLED_CNT);
210 
211     /* 4、创建类 */
212     newchrled.class = class_create(THIS_MODULE, NEWCHRLED_NAME);
213     if (IS_ERR(newchrled.class)) {
214         return PTR_ERR(newchrled.class);
215     }
```

```
216 
217     /* 5、创建设备 */
218     newchrled.device = device_create(newchrled.class, NULL, 
                                   newchrled.devid, NULL, NEWCHRLED_NAME);
219     if (IS_ERR(newchrled.device)) {
220         return PTR_ERR(newchrled.device);
221     }
222 
223     return 0;
224 }
225 
226 /*
227  * @description   : 驱动出口函数
228  * @param         : 无
229  * @return        : 无
230  */
231 static void __exit led_exit(void)
232 {
233     /* 取消映射 */
234     iounmap(IMX6U_CCM_CCGR1);
235     iounmap(SW_MUX_GPIO1_IO03);
236     iounmap(SW_PAD_GPIO1_IO03);
237     iounmap(GPIO1_DR);
238     iounmap(GPIO1_GDIR);
239 
240     /* 注销字符设备 */
241     cdev_del(&newchrled.cdev);/*  删除 cdev */
242     unregister_chrdev_region(newchrled.devid, NEWCHRLED_CNT);
243 
244     device_destroy(newchrled.class, newchrled.devid);
245     class_destroy(newchrled.class);
246 }
247 
248 module_init(led_init);
249 module_exit(led_exit);
250 MODULE_LICENSE("GPL");
251 MODULE_AUTHOR("zuozhongkai");
```

第 25 行，宏 NEWCHRLED_CNT 表示设备数量，在申请设备号或者向 Linux 内核添加字符设备的时候需要设置设备数量，一般我们一个驱动一个设备，所以这个宏为 1。

第 26 行，宏 NEWCHRLED_NAME 表示设备名字，本实验的设备名为“newchrdev”，为了方便管理，所有使用到设备名字的地方统一使用此宏，当驱动加载成功以后就生成/dev/newchrled 这个设备文件。

第 44~52 行，创建设备结构体 newchrled_dev。

第 54 行，定义一个设备结构体变量 newchrdev，此变量表示 led 设备。

第 82~86 行，在 led_open 函数中设置文件的私有数据 private_data 指向 newchrdev。

第 194~221 行，根据前面讲解的方法在驱动入口函数 led_init 中申请设备号、添加字符设备、创建类和设备。本实验我们采用动态申请设备号的方法，第 202 行使用 printk 在终端上显示出申请到的主设备号和次设备号。

第 241~245 行，根据前面讲解的方法，在驱动出口函数 led_exit 中注销字符新设备、删除类和设备。

总体来说 newchrled.c 文件中的内容不复杂，LED 灯驱动部分的程序和上一章一样。重点就是使用了新的字符设备驱动方法。

42.5.2 编写测试 APP

本章直接使用上一章的测试 APP，将上一章的 ledApp.c 文件复制到本章实验工程下即可。

42.6 运行测试

42.6.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 newchrled.o，Makefile 内容如下所示：

示例代码 42.6.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := newchrled.o.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 newchrled.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“newchrled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

42.6.2 运行测试

将上一小节编译出来的 newchrled.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 newchrled.ko 驱动模块：

```
depmod              //第一次加载驱动的时候需要运行此命令
```

```
modprobe newchrled.ko        //加载驱动
```

驱动加载成功以后会输出申请到的主设备号和次设备号，如图 42.6.2.1 所示：

```
/lib/modules/4.1.15 # modprobe newchrled.ko
newcheled major=249,minor=0
/lib/modules/4.1.15 #
```

图 42.6.2.1 申请到的主设备号和次设备号

从图 42.6.2.1 可以看出，申请到的主设备号为 249，次设备号为 0。驱动加载成功以后会自动在/dev 目录下创建设备节点文件/dev/newchrdev，输入如下命令查看/dev/newchrdev 这个设备节点文件是否存在：

```
ls /dev/newchrled -l
```

结果如图 42.6.2.2 所示：

```
/lib/modules/4.1.15 # ls /dev/newchrled -l
crw-rw----    1 0        0         249,   0 Jan  1 05:56 /dev/newchrled
/lib/modules/4.1.15 #
```

图 42.6.2.2 /dev/newchrled 设备节点文件

从图 42.6.2.2 中可以看出，/dev/newchrled 这个设备文件存在，而且主设备号为 249，此设备号为 0，说明设备节点文件创建成功。

驱动节点创建成功以后就可以使用 ledApp 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/newchrled 1        //打开 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否点亮，如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp  /dev/newchrled 0        //关闭 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可：

```
rmmod newchrled.ko
```

第四十三章 Linux 设备树

前面章节中我们多次提到“设备树”这个概念，因为时机未到，所以当时并没有详细的讲解什么是“设备树”，本章我们就来详细的谈一谈设备树。掌握设备树是 Linux 驱动开发人员必备的技能！因为在新版本的 Linux 中，ARM 相关的驱动全部采用了设备树(也有支持老式驱动的，比较少)，最新出的 CPU 其驱动开发也基本都是基于设备树的，比如 ST 新出的 STM32MP157、NXP 的 I.MX8 系列等。我们所使用的 Linux 版本为 4.1.15，其支持设备树，所以正点原子 I.MX6U-ALPHA 开发板的所有 Linux 驱动都是基于设备树的。本章我们就来了解一下设备树的起源、重点学习一下设备树语法。

43.1 什么是设备树？

设备树(Device Tree)，将这个词分开就是"设备"和"树"，描述设备树的文件叫做 DTS(Device Tree Source)，这个 DTS 文件采用树形结构描述板级设备，也就是开发板上的设备信息，比如 CPU 数量、 内存基地址、IIC 接口上接了哪些设备、SPI 接口上接了哪些设备等等，如图 43.1.1 所示：

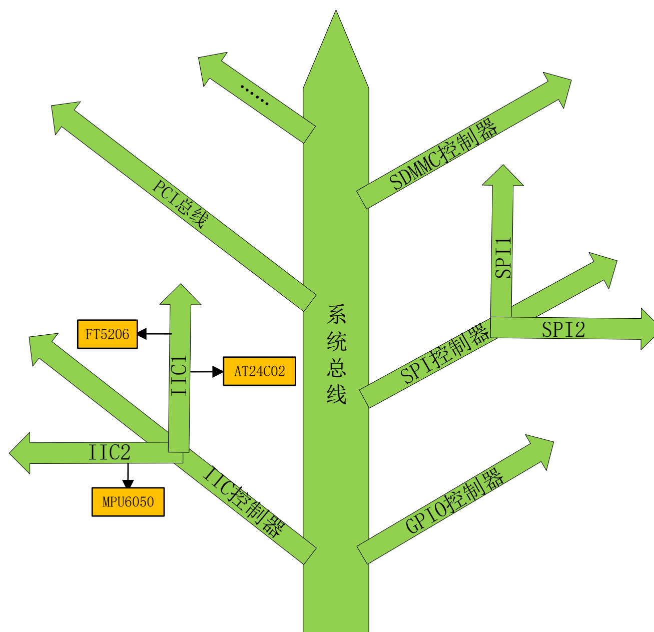


图 43.1.1 设备树结构示意图

在图 43.1.1 中，树的主干就是系统总线，IIC 控制器、GPIO 控制器、SPI 控制器等都是接到系统主线上的分支。IIC 控制器有分为 IIC1 和 IIC2 两种，其中 IIC1 上接了 FT5206 和 AT24C02 这两个 IIC 设备，IIC2 上只接了 MPU6050 这个设备。DTS 文件的主要功能就是按照图 43.1.1 所示的结构来描述板子上的设备信息，DTS 文件描述设备信息是有相应的语法规则要求的，稍后我们会详细的讲解 DTS 语法规则。

在 3.x 版本(具体哪个版本笔者也无从考证)以前的 Linux 内核中 ARM 架构并没有采用设备树。在没有设备树的时候 Linux 是如何描述 ARM 架构中的板级信息呢？在 Linux 内核源码中大量的 arch/arm/mach-xxx 和 arch/arm/plat-xxx 文件夹，这些文件夹里面的文件就是对应平台下的板级信息。比如在 arch/arm/mach-smdk2440.c 中有如下内容(有缩减)：

示例代码 43.1.1 mach-smdk2440.c 文件代码段

```
90  static struct s3c2410fb_display smdk2440_lcd_cfg __initdata = {
91 
92      .lcdcon5    = S3C2410_LCDCON5_FRM565 |
93                S3C2410_LCDCON5_INVVLINE |
94                S3C2410_LCDCON5_INVVFRAME |
95                S3C2410_LCDCON5_PWREN |
96                S3C2410_LCDCON5_HWSWP,
......
113 };
114
```

```
115 static struct s3c2410fb_mach_info smdk2440_fb_info __initdata = {
116     .displays   = &smdk2440_lcd_cfg,
117     .num_displays   = 1,
118     .default_display = 0,
......
133 };
134
135 static struct platform_device *smdk2440_devices[] __initdata = {
136     &s3c_device_ohci,
137     &s3c_device_lcd,
138     &s3c_device_wdt,
139     &s3c_device_i2c0,
140     &s3c_device_iis,
141 };
```

上述代码中的结构体变量 smdk2440_fb_info 就是描述 SMDK2440 这个开发板上的 LCD 信息的，结构体指针数组 smdk2440_devices 描述的 SMDK2440 这个开发板上的所有平台相关信息。这个仅仅是使用 2440 这个芯片的 SMDK2440 开发板下的 LCD 信息，SMDK2440 开发板还有很多的其他外设硬件和平台硬件信息。使用 2440 这个芯片的板子有很多，每个板子都有描述相应板级信息的文件，这仅仅只是一个 2440。随着智能手机的发展，每年新出的 ARM 架构芯片少说都在数十、数百款，Linux 内核下板级信息文件将会成指数级增长！这些板级信息文件都是.c 或.h 文件，都会被硬编码进 Linux 内核中，导致 Linux 内核“虚胖”。就好比你喜欢吃自助餐，然后花了 100 多到一家宣传看着很不错的自助餐厅，结果你想吃的牛排、海鲜、烤肉基本没多少，全都是一些凉菜、炒面、西瓜、饮料等小吃，相信你此时肯定会脱口而出一句“F*k!”、“骗子！”。同样的，当 Linux 之父 linus 看到 ARM 社区向 Linux 内核添加了大量“无用”、冗余的板级信息文件，不禁的发出了一句“This whole ARM thing is a f*cking pain in the ass”。从此以后 ARM 社区就引入了 PowerPC 等架构已经采用的设备树(Flattened Device Tree)，将这些描述板级硬件信息的内容都从 Linux 内中分离开来，用一个专属的文件格式来描述，这个专属的文件就叫做设备树，文件扩展名为.dts。一个 SOC 可以作出很多不同的板子，这些不同的板子肯定是有共同的信息，将这些共同的信息提取出来作为一个通用的文件，其他的.dts 文件直接引用这个通用文件即可，这个通用文件就是.dtsi 文件，类似于 C 语言中的头文件。一般.dts 描述板级信息(也就是开发板上有哪些 IIC 设备、SPI 设备等)，.dtsi 描述 SOC 级信息(也就是 SOC 有几个 CPU、主频是多少、各个外设控制器信息等)。

这个就是设备树的由来，简而言之就是，Linux 内核中 ARM 架构下有太多的冗余的垃圾板级信息文件，导致 linus 震怒，然后 ARM 社区引入了设备树。

43.2 DTS、DTB 和 DTC

上一小节说了，设备树源文件扩展名为.dts，但是我们在前面移植 Linux 的时候却一直在使用.dtb 文件，那么 DTS 和 DTB 这两个文件是什么关系呢？DTS 是设备树源码文件，DTB 是将 DTS 编译以后得到的二进制文件。将.c 文件编译为.o 需要用到 gcc 编译器，那么将.dts 编译为.dtb 需要什么工具呢？需要用到 DTC 工具！DTC 工具源码在 Linux 内核的 scripts/dtc 目录下，scripts/dtc/Makefile 文件内容如下：

示例代码 43.2.1 scripts/dtc/Makefile 文件代码段

```
1 hostprogs-y := dtc
```

```
2 always       := $(hostprogs-y)
3
4 dtc-objs:= dtc.o flattree.o fstree.o data.o livetree.o treesource.o \
5          srcpos.o checks.o util.o
6 dtc-objs  += dtc-lexer.lex.o dtc-parser.tab.o
......
```

可以看出，DTC 工具依赖于 dtc.c、flattree.c、fstree.c 等文件，最终编译并链接出 DTC 这个主机文件。如果要编译 DTS 文件的话只需要进入到 Linux 源码根目录下，然后执行如下命令：

```
make all
```

或者：

```
make dtbs
```

“make all”命令是编译 Linux 源码中的所有东西，包括 zImage，.ko 驱动模块以及设备树，如果只是编译设备树的话建议使用“make dtbs”命令。

基于 ARM 架构的 SOC 有很多种，一种 SOC 又可以制作出很多款板子，每个板子都有一个对应的 DTS 文件，那么如何确定编译哪一个 DTS 文件呢？我们就以 I.MX6ULL 这款芯片对应的板子为例来看一下，打开 arch/arm/boot/dts/Makefile，有如下内容：

示例代码 43.2.2 arch/arm/boot/dts/Makefile 文件代码段

```
381 dtb-$(CONFIG_SOC_IMX6UL) += \
382     imx6ul-14x14-ddr3-arm2.dtb \
383     imx6ul-14x14-ddr3-arm2-emmc.dtb \
......
400 dtb-$(CONFIG_SOC_IMX6ULL) += \
401     imx6ull-14x14-ddr3-arm2.dtb \
402     imx6ull-14x14-ddr3-arm2-adc.dtb \
403     imx6ull-14x14-ddr3-arm2-cs42888.dtb \
404     imx6ull-14x14-ddr3-arm2-ecspi.dtb \
405     imx6ull-14x14-ddr3-arm2-emmc.dtb \
406     imx6ull-14x14-ddr3-arm2-epdc.dtb \
407     imx6ull-14x14-ddr3-arm2-flexcan2.dtb \
408     imx6ull-14x14-ddr3-arm2-gpmi-weim.dtb \
409     imx6ull-14x14-ddr3-arm2-lcdif.dtb \
410     imx6ull-14x14-ddr3-arm2-ldo.dtb \
411     imx6ull-14x14-ddr3-arm2-qspi.dtb \
412     imx6ull-14x14-ddr3-arm2-qspi-all.dtb \
413     imx6ull-14x14-ddr3-arm2-tsc.dtb \
414     imx6ull-14x14-ddr3-arm2-uart2.dtb \
415     imx6ull-14x14-ddr3-arm2-usb.dtb \
416     imx6ull-14x14-ddr3-arm2-wm8958.dtb \
417     imx6ull-14x14-evk.dtb \
418     imx6ull-14x14-evk-btwifi.dtb \
419     imx6ull-14x14-evk-emmc.dtb \
420     imx6ull-14x14-evk-gpmi-weim.dtb \
```

```
421     imx6ull-14x14-evk-usb-certi.dtb \
422     imx6ull-alientek-emmc.dtb \
423     imx6ull-alientek-nand.dtb \
424     imx6ull-9x9-evk.dtb \
425     imx6ull-9x9-evk-btwifi.dtb \
426     imx6ull-9x9-evk-ldo.dtb
427 dtb-$(CONFIG_SOC_IMX6SLL) += \
428     imx6sll-lpddr2-arm2.dtb \
429     imx6sll-lpddr3-arm2.dtb \
......
```

可以看出，当选中 I.MX6ULL 这个 SOC 以后(CONFIG_SOC_IMX6ULL=y)，所有使用到 I.MX6ULL 这个 SOC 的板子对应的.dts 文件都会被编译为.dtb。如果我们使用 I.MX6ULL 新做了一个板子，只需要新建一个此板子对应的.dts 文件，然后将对应的.dtb 文件名添加到 dtb-$(CONFIG_SOC_IMX6ULL)下，这样在编译设备树的时候就会将对应的.dts 编译为二进制的.dtb 文件。

示例代码 43.2.2 中第 422 和 423 行就是我们在给正点原子的 I.MX6U-ALPHA 开发板移植 Linux 系统的时候添加的设备树。关于.dtb 文件怎么使用这里就不多说了，前面讲解 Uboot 移植、Linux 内核移植的时候已经无数次的提到如何使用.dtb 文件了(uboot 中使用 bootz 或 bootm 命令向 Linux 内核传递二进制设备树文件(.dtb))。

43.3 DTS 语法

虽然我们基本上不会从头到尾重写一个.dts 文件，大多时候是直接在 SOC 厂商提供的.dts 文件上进行修改。但是 DTS 文件语法我们还是需要详细的学习一遍，因为我们肯定需要修改.dts 文件。大家不要看到要学习新的语法就觉得会很复杂，DTS 语法非常的人性化，是一种 ASCII 文本文件，不管是阅读还是修改都很方便。

本节我们就以 imx6ull-alientek-emmc.dts 这个文件为例来讲解一下 DTS 语法。关于设备树详细的语法规则请参考《Devicetree SpecificationV0.2.pdf》和《Power_ePAPR_APPROVED_v1.12.pdf》这两份文档，此两份文档已经放到了开发板光盘中，路径为：4、参考资料->Devicetree SpecificationV0.2.pdf、4、参考资料->Power_ePAPR_APPROVED_v1.12.pdf

43.3.1 .dtsi 头文件

和 C 语言一样，设备树也支持头文件，设备树的头文件扩展名为.dtsi。在 imx6ull-alientek-emmc.dts 中有如下所示内容：

示例代码 43.3.1.1 imx6ull-alientek-emmc.dts 文件代码段

```
12 #include <dt-bindings/input/input.h>
13 #include "imx6ull.dtsi"
```

第 12 行，使用“#include”来引用“input.h”这个.h 头文件。

第 13 行，使用“#include”来引用“imx6ull.dtsi”这个.dtsi 头文件。

看到这里，大家可能会疑惑，不是说设备树的扩展名是.dtsi 吗？为什么也可以直接引用 C 语言中的.h 头文件呢？这里并没有错，.dts 文件引用 C 语言中的.h 文件，甚至也可以引用.dts 文件，打开 imx6ull-14x14-evk-gpmi-weim.dts 这个文件，此文件中有如下内容：

示例代码 43.3.1.2 imx6ull-14x14-evk-gpmi-weim.dts 文件代码段

```
9 #include "imx6ull-14x14-evk.dts"
```

可以看出，示例代码 43.3.1.2 中直接引用了.dts 文件，因此在.dts 设备树文件中，可以通过“#include”来引用.h、.dtsi 和.dts 文件。只是，我们在编写设备树头文件的时候最好选择.dtsi 后缀。

一般.dtsi 文件用于描述 SOC 的内部外设信息，比如 CPU 架构、主频、外设寄存器地址范围，比如 UART、IIC 等等。比如 imx6ull.dtsi 就是描述 I.MX6ULL 这颗 SOC 内部外设情况信息的，内容如下：

示例代码 43.3.1.3 imx6ull.dtsi 文件代码段

```
10  #include <dt-bindings/clock/imx6ul-clock.h>
11  #include <dt-bindings/gpio/gpio.h>
12  #include <dt-bindings/interrupt-controller/arm-gic.h>
13  #include "imx6ull-pinfunc.h"
14  #include "imx6ull-pinfunc-snvs.h"
15  #include "skeleton.dtsi"
16
17  / {
18      aliases {
19          can0 = &flexcan1;
......
48      };
49
50      cpus {
51          #address-cells = <1>;
52          #size-cells = <0>;
53
54          cpu0: cpu@0 {
55              compatible = "arm,cortex-a7";
56              device_type = "cpu";
......
89          };
90      };
91
92      intc: interrupt-controller@00a01000 {
93          compatible = "arm,cortex-a7-gic";
94          #interrupt-cells = <3>;
95          interrupt-controller;
96          reg = <0x00a01000 0x1000>,
97                <0x00a02000 0x100>;
98      };
99
100     clocks {
101         #address-cells = <1>;
```

```
102         #size-cells = <0>;
103 
104         ckil: clock@0 {
105             compatible = "fixed-clock";
106             reg = <0>;
107             #clock-cells = <0>;
108             clock-frequency = <32768>;
109             clock-output-names = "ckil";
110         };
......
135     };
136 
137     soc {
138         #address-cells = <1>;
139         #size-cells = <1>;
140         compatible = "simple-bus";
141         interrupt-parent = <&gpc>;
142         ranges;
143 
144         busfreq {
145             compatible = "fsl,imx_busfreq";
......
162         };
197 
198         gpmi: gpmi-nand@01806000{
199             compatible = "fsl,imx6ull-gpmi-nand", "fsl, imx6ul-gpmi-
nand";
200             #address-cells = <1>;
201             #size-cells = <1>;
202             reg = <0x01806000 0x2000>, <0x01808000 0x4000>;
......
216         };
......
1177    };
1178 };
```

示例代码 43.3.1.3 中第 54~89 行就是 cpu0 这个设备节点信息，这个节点信息描述了 I.MX6ULL 这颗 SOC 所使用的 CPU 信息，比如架构是 cortex-A7，频率支持 996MHz、792MHz、528MHz、396MHz 和 198MHz 等等。在 imx6ull.dtsi 文件中不仅仅描述了 cpu0 这一个节点信息，I.MX6ULL 这颗 SOC 所有的外设都描述的清清楚楚，比如 ecspi1~4、uart1~8、usbphy1~2、i2c1~4 等等，关于这些设备节点信息的具体内容我们稍后在详细的讲解。

43.3.2 设备节点

设备树是采用树形结构来描述板子上的设备信息的文件，每个设备都是一个节点，叫做设备节点，每个节点都通过一些属性信息来描述节点信息，属性就是键—值对。以下是从 imx6ull.dtsi 文件中缩减出来的设备树文件内容：

示例代码 43.3.2.1 设备树模板

```
1  / {
2      aliases {
3          can0 = &flexcan1;
4      };
5  
6      cpus {
7          #address-cells = <1>;
8          #size-cells = <0>;
9  
10         cpu0: cpu@0 {
11             compatible = "arm,cortex-a7";
12             device_type = "cpu";
13             reg = <0>;
14         };
15     };
16 
17     intc: interrupt-controller@00a01000 {
18         compatible = "arm,cortex-a7-gic";
19         #interrupt-cells = <3>;
20         interrupt-controller;
21         reg = <0x00a01000 0x1000>,
22               <0x00a02000 0x100>;
23     };
24 }
```

第 1 行，“/”是根节点，每个设备树文件只有一个根节点。细心的同学应该会发现，imx6ull.dtsi 和 imx6ull-alientek-emmc.dts 这两个文件都有一个“/”根节点，这样不会出错吗？不会的，因为这两个“/”根节点的内容会合并成一个根节点。

第 2、6 和 17 行，aliases、cpus 和 intc 是三个子节点，在设备树中节点命名格式如下：

```
node-name@unit-address
```

其中“node-name”是节点名字，为 ASCII 字符串，节点名字应该能够清晰的描述出节点的功能，比如“uart1”就表示这个节点是 UART1 外设。“unit-address”一般表示设备的地址或寄存器首地址，如果某个节点没有地址或者寄存器的话“unit-address”可以不要，比如“cpu@0”、“interrupt-controller@00a01000”。

但是我们在示例代码 43.3.2.1 中我们看到的节点命名却如下所示：

```
cpu0:cpu@0
```

上述命令并不是“node-name@unit-address”这样的格式，而是用“：”隔开成了两部分，“：”前面的是节点标签(label)，“：”后面的才是节点名字，格式如下所示：

```
label: node-name@unit-address
```

引入 label 的目的就是为了方便访问节点，可以直接通过&label 来访问这个节点，比如通过&cpu0 就可以访问"cpu@0"这个节点，而不需要输入完整的节点名字。再比如节点 "intc: interrupt-controller@00a01000"，节点 label 是 intc，而节点名字就很长了，为"interrupt-controller@00a01000"。很明显通过&intc 来访问"interrupt-controller@00a01000"这个节点要方便很多！

第 10 行，cpu0 也是一个节点，只是 cpu0 是 cpus 的子节点。

每个节点都有不同属性，不同的属性又有不同的内容，属性都是键值对，值可以为空或任意的字节流。设备树源码中常用的几种数据形式如下所示：

①、字符串

```
compatible = "arm,cortex-a7";
```

上述代码设置 compatible 属性的值为字符串"arm,cortex-a7"。

②、32 位无符号整数

```
reg = <0>;
```

上述代码设置 reg 属性的值为 0，reg 的值也可以设置为一组值，比如：

```
reg = <0 0x123456 100>;
```

③、字符串列表

属性值也可以为字符串列表，字符串和字符串之间采用","隔开，如下所示：

```
compatible = "fsl,imx6ull-gpmi-nand", "fsl, imx6ul-gpmi-nand";
```

上述代码设置属性 compatible 的值为"fsl,imx6ull-gpmi-nand"和"fsl, imx6ul-gpmi-nand"。

43.3.3 标准属性

节点是由一堆的属性组成，节点都是具体的设备，不同的设备需要的属性不同，用户可以自定义属性。除了用户自定义属性，有很多属性是标准属性，Linux 下的很多外设驱动都会使用这些标准属性，本节我们就来学习一下几个常用的标准属性。

1、compatible 属性

compatible 属性也叫做"兼容性"属性，这是非常重要的一个属性！compatible 属性的值是一个字符串列表，compatible 属性用于将设备和驱动绑定起来。字符串列表用于选择设备所要使用的驱动程序，compatible 属性的值格式如下所示：

```
"manufacturer,model"
```

其中 manufacturer 表示厂商，model 一般是模块对应的驱动名字。比如 imx6ull-alientek-emmc.dts 中 sound 节点是 I.MX6U-ALPHA 开发板的音频设备节点，I.MX6U-ALPHA 开发板上的音频芯片采用的欧胜(WOLFSON)出品的 WM8960，sound 节点的 compatible 属性值如下：

```
compatible = "fsl,imx6ul-evk-wm8960","fsl,imx-audio-wm8960";
```

属性值有两个，分别为"fsl,imx6ul-evk-wm8960"和"fsl,imx-audio-wm8960"，其中"fsl"表示厂商是飞思卡尔，"imx6ul-evk-wm8960"和"imx-audio-wm8960"表示驱动模块名字。sound 这个设备首先使用第一个兼容值在 Linux 内核里面查找，看看能不能找到与之匹配的驱动文件，如果没有找到的话就使用第二个兼容值查找，直到找到或者查找完整个 Linux 内核也没有找到对应的驱动。

一般驱动程序文件都会有一个 OF 匹配表，此 OF 匹配表保存着一些 compatible 值，如果设备节点的 compatible 属性值和 OF 匹配表中的任何一个值相等，那么就表示设备可以使用这个驱动。比如在文件 imx-wm8960.c 中有如下内容：

示例代码 43.3.3.1 imx-wm8960.c 文件代码段

```
632 static const struct of_device_id imx_wm8960_dt_ids[] = {
633     { .compatible = "fsl,imx-audio-wm8960", },
634     { /* sentinel */ }
635 };
636 MODULE_DEVICE_TABLE(of, imx_wm8960_dt_ids);
637
638 static struct platform_driver imx_wm8960_driver = {
639     .driver = {
640         .name = "imx-wm8960",
641         .pm = &snd_soc_pm_ops,
642         .of_match_table = imx_wm8960_dt_ids,
643     },
644     .probe = imx_wm8960_probe,
645     .remove = imx_wm8960_remove,
646 };
```

第 632~635 行的数组 imx_wm8960_dt_ids 就是 imx-wm8960.c 这个驱动文件的匹配表，此匹配表只有一个匹配值“fsl,imx-audio-wm8960”。如果在设备树中有哪个节点的 compatible 属性值与此相等，那么这个节点就会使用此驱动文件。

第 642 行，wm8960 采用了 platform_driver 驱动模式，关于 platform_driver 驱动后面会讲解。此行设置.of_match_table 为 imx_wm8960_dt_ids，也就是设置这个 platform_driver 所使用的 OF 匹配表。

2、model 属性

model 属性值也是一个字符串，一般 model 属性描述设备模块信息，比如名字什么的，比如：

```
model = "wm8960-audio";
```

3、status 属性

status 属性看名字就知道是和设备状态有关的，status 属性值也是字符串，字符串是设备的状态信息，可选的状态如表 43.3.3.1 所示：

值	描述
“okay”	表明设备是可操作的。
“disabled”	表明设备当前是不可操作的，但是在未来可以变为可操作的，比如热插拔设备插入以后。至于 disabled 的具体含义还要看设备的绑定文档。
“fail”	表明设备不可操作，设备检测到了一系列的错误，而且设备也不大可能变得可操作。
“fail-sss”	含义和“fail”相同，后面的 sss 部分是检测到的错误内容。

表 43.3.3.1 status 属性值表

4、#address-cells 和#size-cells 属性

这两个属性的值都是无符号 32 位整形，#address-cells 和#size-cells 这两个属性可以用在任何拥有子节点的设备中，用于描述子节点的地址信息。#address-cells 属性值决定了子节点 reg 属性中地址信息所占用的字长(32 位)，#size-cells 属性值决定了子节点 reg 属性中长度信息所占的

字长(32 位)。#address-cells 和#size-cells 表明了子节点应该如何编写 reg 属性值，一般 reg 属性都是和地址有关的内容，和地址相关的信息有两种：起始地址和地址长度，reg 属性的格式一为：

```
reg = <address1 length1 address2 length2 address3 length3……>
```

每个“address length”组合表示一个地址范围，其中 address 是起始地址，length 是地址长度，#address-cells 表明 address 这个数据所占用的字长，#size-cells 表明 length 这个数据所占用的字长，比如:

示例代码 43.3.3.2 #address-cells 和#size-cells 属性

```
1  spi4 {
2      compatible = "spi-gpio";
3      #address-cells = <1>;
4      #size-cells = <0>;
5  
6      gpio_spi: gpio_spi@0 {
7          compatible = "fairchild,74hc595";
8          reg = <0>;
9      };
10 };
11 
12 aips3: aips-bus@02200000 {
13     compatible = "fsl,aips-bus", "simple-bus";
14     #address-cells = <1>;
15     #size-cells = <1>;
16 
17     dcp: dcp@02280000 {
18         compatible = "fsl,imx6sl-dcp";
19         reg = <0x02280000 0x4000>;
20     };
21 };
```

第 2，3 行，节点 spi4 的#address-cells = <1>，#size-cells = <0>，说明 spi4 的子节点 reg 属性中起始地址所占用的字长为 1，地址长度所占用的字长为 0。

第 8 行，子节点 gpio_spi: gpio_spi@0 的 reg 属性值为 <0>，因为父节点设置了#address-cells = <1>，#size-cells = <0>，因此 addres=0，没有 length 的值，相当于设置了起始地址，而没有设置地址长度。

第 14，15 行，设置 aips3: aips-bus@02200000 节点#address-cells = <1>，#size-cells = <1>，说明 aips3: aips-bus@02200000 节点起始地址长度所占用的字长为 1，地址长度所占用的字长也为 1。

第 19 行，子节点 dcp: dcp@02280000 的 reg 属性值为<0x02280000 0x4000>，因为父节点设置了#address-cells = <1>，#size-cells = <1>，address= 0x02280000，length= 0x4000，相当于设置了起始地址为 0x02280000，地址长度为 0x40000。

5、reg 属性

reg 属性前面已经提到过了，reg 属性的值一般是(address，length)对。reg 属性一般用于描述设备地址空间资源信息，一般都是某个外设的寄存器地址范围信息，比如在 imx6ull.dtsi 中有如下内容：

示例代码 43.3.3.3 uart1 节点信息

```
323 uart1: serial@02020000 {
324     compatible = "fsl,imx6ul-uart",
325             "fsl,imx6q-uart", "fsl,imx21-uart";
326     reg = <0x02020000 0x4000>;
327     interrupts = <GIC_SPI 26 IRQ_TYPE_LEVEL_HIGH>;
328     clocks = <&clks IMX6UL_CLK_UART1_IPG>,
329         <&clks IMX6UL_CLK_UART1_SERIAL>;
330     clock-names = "ipg", "per";
331     status = "disabled";
332 };
```

上述代码是节点 uart1，uart1 节点描述了 I.MX6ULL 的 UART1 相关信息，重点是第 326 行的 reg 属性。其中 uart1 的父节点 aips1: aips-bus@02000000 设置了#address-cells = <1>、#size-cells = <1>，因此 reg 属性中 address=0x02020000，length=0x4000。查阅《I.MX6ULL 参考手册》可知，I.MX6ULL 的 UART1 寄存器首地址为 0x02020000，但是 UART1 的地址长度(范围)并没有 0x4000 这么多，这里我们重点是获取 UART1 寄存器首地址。

6、ranges 属性

ranges 属性值可以为空或者按照(child-bus-address,parent-bus-address,length)格式编写的数字矩阵，ranges 是一个地址映射/转换表，ranges 属性每个项目由子地址、父地址和地址空间长度这三部分组成：

child-bus-address：子总线地址空间的物理地址，由父节点的#address-cells 确定此物理地址所占用的字长。

parent-bus-address：父总线地址空间的物理地址，同样由父节点的#address-cells 确定此物理地址所占用的字长。

length：子地址空间的长度，由父节点的#size-cells 确定此地址长度所占用的字长。

如果 ranges 属性值为空值，说明子地址空间和父地址空间完全相同，不需要进行地址转换，对于我们所使用的 I.MX6ULL 来说，子地址空间和父地址空间完全相同，因此会在 imx6ull.dtsi 中找到大量的值为空的 ranges 属性，如下所示：

示例代码 43.3.3.4 imx6ull.dtsi 文件代码段

```
137 soc {
138     #address-cells = <1>;
139     #size-cells = <1>;
140     compatible = "simple-bus";
141     interrupt-parent = <&gpc>;
142     ranges;
......
1177 }
```

第 142 行定义了 ranges 属性，但是 ranges 属性值为空。

ranges 属性不为空的示例代码如下所示：

示例代码 43.3.3.5 ranges 属性不为空

```
1  soc {
2      compatible = "simple-bus";
3      #address-cells = <1>;
```

```
4       #size-cells = <1>;
5       ranges = <0x0 0xe0000000 0x00100000>;
6
7       serial {
8           device_type = "serial";
9           compatible = "ns16550";
10          reg = <0x4600 0x100>;
11          clock-frequency = <0>;
12          interrupts = <0xA 0x8>;
13          interrupt-parent = <&ipic>;
14      };
15 };
```

第 5 行，节点 soc 定义的 ranges 属性，值为<0x0 0xe0000000 0x00100000>，此属性值指定了一个 1024KB(0x00100000)的地址范围，子地址空间的物理起始地址为 0x0，父地址空间的物理起始地址为 0xe0000000。

第 6 行，serial 是串口设备节点，reg 属性定义了 serial 设备寄存器的起始地址为 0x4600，寄存器长度为 0x100。经过地址转换，serial 设备可以从 0xe0004600 开始进行读写操作，0xe0004600=0x4600+0xe0000000。

7、name 属性

name 属性值为字符串，name 属性用于记录节点名字，name 属性已经被弃用，不推荐使用 name 属性，一些老的设备树文件可能会使用此属性。

8、device_type 属性

device_type 属性值为字符串，IEEE 1275 会用到此属性，用于描述设备的 FCode，但是设备树没有 FCode，所以此属性也被抛弃了。此属性只能用于 cpu 节点或者 memory 节点。imx6ull.dtsi 的 cpu0 节点用到了此属性，内容如下所示：

示例代码 43.3.3.6 imx6ull.dtsi 文件代码段

```
54 cpu0: cpu@0 {
55      compatible = "arm,cortex-a7";
56      device_type = "cpu";
57      reg = <0>;
......
89 };
```

关于标准属性就讲解这么多，其他的比如中断、IIC、SPI 等使用的标准属性等到具体的例程在讲解。

43.3.4 根节点 compatible 属性

每个节点都有 compatible 属性，根节点“/”也不例外，imx6ull-alientek-emmc.dts 文件中根节点的 compatible 属性内容如下所示：

示例代码 43.3.4.1 imx6ull-alientek-emmc.dts 根节点 compatible 属性

```
14 / {
15      model = "Freescale i.MX6 ULL 14x14 EVK Board";
16      compatible = "fsl,imx6ull-14x14-evk", "fsl,imx6ull";
```

```
......
148 }
```

可以看出，compatible 有两个值："fsl,imx6ull-14x14-evk"和"fsl,imx6ull"。前面我们说了，设备节点的 compatible 属性值是为了匹配 Linux 内核中的驱动程序，那么根节点中的 compatible 属性是为了做什么工作的？ 通过根节点的 compatible 属性可以知道我们所使用的设备，一般第一个值描述了所使用的硬件设备名字，比如这里使用的是"imx6ull-14x14-evk"这个设备，第二个值描述了设备所使用的 SOC，比如这里使用的是"imx6ull"这颗 SOC。Linux 内核会通过根节点的 compoatible 属性查看是否支持此设备，如果支持的话设备就会启动 Linux 内核。接下来我们就来学习一下 Linux 内核在使用设备树前后是如何判断是否支持某款设备的。

1、使用设备树之前设备匹配方法

在没有使用设备树以前，uboot 会向 Linux 内核传递一个叫做 machine id 的值，machine id 也就是设备 ID，告诉 Linux 内核自己是个什么设备，看看 Linux 内核是否支持。Linux 内核是支持很多设备的，针对每一个设备(板子)，Linux 内核都用 MACHINE_START 和 MACHINE_END 来定义一个 machine_desc 结构体来描述这个设备，比如在文件 arch/arm/mach-imx/mach-mx35_3ds.c 中有如下定义：

示例代码 43.3.4.2 MX35_3DS 设备

```
613 MACHINE_START(MX35_3DS, "Freescale MX35PDK")
614     /* Maintainer: Freescale Semiconductor, Inc */
615     .atag_offset = 0x100,
616     .map_io = mx35_map_io,
617     .init_early = imx35_init_early,
618     .init_irq = mx35_init_irq,
619     .init_time  = mx35pdk_timer_init,
620     .init_machine = mx35_3ds_init,
621     .reserve = mx35_3ds_reserve,
622     .restart    = mxc_restart,
623 MACHINE_END
```

上述代码就是定义了"Freescale MX35PDK"这个设备，其中 MACHINE_START 和 MACHINE_END 定义在文件 arch/arm/include/asm/mach/arch.h 中，内容如下：

示例代码 43.3.4.3 MACHINE_START 和 MACHINE_END 宏定义

```
#define MACHINE_START(_type,_name)          \
static const struct machine_desc __mach_desc_##_type    \
 __used                     \
 __attribute__((__section__(".arch.info.init"))) = {    \
    .nr     = MACH_TYPE_##_type,        \
    .name       = _name,

#define MACHINE_END             \
};
```

根据 MACHINE_START 和 MACHINE_END 的宏定义，将示例代码 43.3.4.2 展开后如下所示：

示例代码 43.3.4.3 展开以后

```
1  static const struct machine_desc __mach_desc_MX35_3DS    \
```

```
2       __used                                              \
3       __attribute__((__section__(".arch.info.init"))) = {
4       .nr     = MACH_TYPE_MX35_3DS,
5       .name       = "Freescale MX35PDK",
6       /* Maintainer: Freescale Semiconductor, Inc */
7       .atag_offset = 0x100,
8       .map_io = mx35_map_io,
9       .init_early = imx35_init_early,
10      .init_irq = mx35_init_irq,
11      .init_time  = mx35pdk_timer_init,
12      .init_machine = mx35_3ds_init,
13      .reserve = mx35_3ds_reserve,
14      .restart    = mxc_restart,
15 }
```

从示例代码 43.3.4.3 中可以看出，这里定义了一个 machine_desc 类型的结构体变量 __mach_desc_MX35_3DS，这个变量存储在“.arch.info.init”段中。第 4 行的 MACH_TYPE_MX35_3DS 就是“Freescale MX35PDK”这个板子的 machine id。MACH_TYPE_MX35_3DS 定义在文件 include/generated/mach-types.h 中，此文件定义了大量的 machine id，内容如下所示：

示例代码 43.3.4.3 mach-types.h 文件中的 machine id

```
15   #define MACH_TYPE_EBSA110              0
16   #define MACH_TYPE_RISCPC               1
17   #define MACH_TYPE_EBSA285              4
18   #define MACH_TYPE_NETWINDER            5
19   #define MACH_TYPE_CATS                 6
20   #define MACH_TYPE_SHARK                15
21   #define MACH_TYPE_BRUTUS               16
22   #define MACH_TYPE_PERSONAL_SERVER      17
......
287  #define MACH_TYPE_MX35_3DS             1645
......
1000 #define MACH_TYPE_PFLA03               4575
```

第 287 行就是 MACH_TYPE_MX35_3DS 的值，为 1645。

前面说了，uboot 会给 Linux 内核传递 machine id 这个参数，Linux 内核会检查这个 machine id，其实就是将 machine id 与示例代码 43.3.4.3 中的这些 MACH_TYPE_XXX 宏进行对比，看看有没有相等的，如果相等的话就表示 Linux 内核支持这个设备，如果不支持的话那么这个设备就没法启动 Linux 内核。

2、使用设备树以后的设备匹配方法

当 Linux 内核引入设备树以后就不再使用 MACHINE_START 了，而是换为了 DT_MACHINE_START。DT_MACHINE_START 也定义在文件 arch/arm/include/asm/mach/arch.h 里面，定义如下：

示例代码 43.3.4.4 DT_MACHINE_START 宏

```
#define DT_MACHINE_START(_name, _namestr)              \
```

```
static const struct machine_desc __mach_desc_##_name     \
 __used                                                   \
 __attribute__((__section__(".arch.info.init"))) = {     \
    .nr     = ~0,                                         \
    .name       = _namestr,
```

可以看出，DT_MACHINE_START 和 MACHINE_START 基本相同，只是.nr 的设置不同，在 DT_MACHINE_START 里面直接将.nr 设置为~0。说明引入设备树以后不会再根据 machine id 来检查 Linux 内核是否支持某个设备了。

打开文件 arch/arm/mach-imx/mach-imx6ul.c，有如下所示内容：

示例代码 43.3.4.5 imx6ull 设备

```
208 static const char *imx6ul_dt_compat[] __initconst = {
209     "fsl,imx6ul",
210     "fsl,imx6ull",
211     NULL,
212 };
213
214 DT_MACHINE_START(IMX6UL, "Freescale i.MX6 Ultralite (Device Tree)")
215     .map_io     = imx6ul_map_io,
216     .init_irq   = imx6ul_init_irq,
217     .init_machine   = imx6ul_init_machine,
218     .init_late  = imx6ul_init_late,
219     .dt_compat  = imx6ul_dt_compat,
220 MACHINE_END
```

machine_desc 结构体中有个.dt_compat 成员变量，此成员变量保存着本设备兼容属性，示例代码 43.3.4.5 中设置.dt_compat ＝imx6ul_dt_compat，imx6ul_dt_compat 表里面有"fsl,imx6ul"和"fsl,imx6ull"这两个兼容值。只要某个设备(板子)根节点“/”的 compatible 属性值与 imx6ul_dt_compat 表中的任何一个值相等，那么就表示 Linux 内核支持此设备。imx6ull-alientek-emmc.dts 中根节点的 compatible 属性值如下：

```
compatible = "fsl,imx6ull-14x14-evk", "fsl,imx6ull";
```

其中“fsl,imx6ull”与 imx6ul_dt_compat 中的“fsl,imx6ull”匹配，因此 I.MX6U-ALPHA 开发板可以正常启动 Linux 内核。如果将 imx6ull-alientek-emmc.dts 根节点的 compatible 属性改为其他的值，比如：

```
compatible = "fsl,imx6ull-14x14-evk", "fsl,imx6ullll"
```

重新编译 DTS，并用新的 DTS 启动 Linux 内核，结果如图 43.3.4.1 所示的错误提示：

```
done
Bytes transferred = 36298 (8dca hex)
Kernel image @ 0x80800000 [ 0x000000 - 0x54a1e0 ]
## Flattened Device Tree blob at 83000000
   Booting using the fdt blob at 0x83000000
   Using Device Tree in place at 83000000, end 8300bdc9

Starting kernel ...
```

输出Starting kernel...以后再无任何信息输出，
Linux Kernel启动失败。

图 43.3.4.1 系统启动信息

当我们修改了根节点 compatible 属性内容以后，因为 Linux 内核找不到对应的设备，因此 Linux 内核无法启动。在 uboot 输出 Starting kernel…以后就再也没有其他信息输出了。

接下来我们简单看一下 Linux 内核是如何根据设备树根节点的 compatible 属性来匹配出对应的 machine_desc，Linux 内核调用 start_kernel 函数来启动内核，start_kernel 函数会调用 setup_arch 函数来匹配 machine_desc，setup_arch 函数定义在文件 arch/arm/kernel/setup.c 中，函数内容如下(有缩减)：

示例代码 43.3.4.6 setup_arch 函数内容

```
913 void __init setup_arch(char **cmdline_p)
914 {
915     const struct machine_desc *mdesc;
916
917     setup_processor();
918     mdesc = setup_machine_fdt(__atags_pointer);
919     if (!mdesc)
920         mdesc = setup_machine_tags(__atags_pointer,
                                        __machine_arch_type);
921     machine_desc = mdesc;
922     machine_name = mdesc->name;
......
986 }
```

第 918 行，调用 setup_machine_fdt 函数来获取匹配的 machine_desc，参数就是 atags 的首地址，也就是 uboot 传递给 Linux 内核的 dtb 文件首地址，setup_machine_fdt 函数的返回值就是找到的最匹配的 machine_desc。

函数 setup_machine_fdt 定义在文件 arch/arm/kernel/devtree.c 中，内容如下(有缩减)：

示例代码 43.3.4.7 setup_machine_fdt 函数内容

```
204 const struct machine_desc * __init setup_machine_fdt(unsigned int
                                                          dt_phys)
205 {
206     const struct machine_desc *mdesc, *mdesc_best = NULL;
......
214
215     if (!dt_phys || !early_init_dt_verify(phys_to_virt(dt_phys)))
216         return NULL;
217
218     mdesc = of_flat_dt_match_machine(mdesc_best,
                                         arch_get_next_mach);
219
......
247     __machine_arch_type = mdesc->nr;
248
249     return mdesc;
250 }
```

第 218 行，调用函数 of_flat_dt_match_machine 来获取匹配的 machine_desc，参数 mdesc_best 是默认的 machine_desc，参数 arch_get_next_mach 是个函数，此函数定义在定义在 arch/arm/kernel/devtree.c 文件中。找到匹配的 machine_desc 的过程就是用设备树根节点的 compatible 属性值和 Linux 内核中保存的所以 machine_desc 结构的. dt_compat 中的值比较，看看那个相等，如果相等的话就表示找到匹配的 machine_desc，arch_get_next_mach 函数的工作就是获取 Linux 内核中下一个 machine_desc 结构体。

最后在来看一下 of_flat_dt_match_machine 函数，此函数定义在文件 drivers/of/fdt.c 中，内容如下(有缩减)：

示例代码 43.3.4.8 of_flat_dt_match_machine 函数内容

```
705 const void * __init of_flat_dt_match_machine(const void 
*default_match,
706         const void * (*get_next_compat)(const char * const**))
707 {
708     const void *data = NULL;
709     const void *best_data = default_match;
710     const char *const *compat;
711     unsigned long dt_root;
712     unsigned int best_score = ~1, score = 0;
713 
714     dt_root = of_get_flat_dt_root();
715     while ((data = get_next_compat(&compat))) {
716         score = of_flat_dt_match(dt_root, compat);
717         if (score > 0 && score < best_score) {
718             best_data = data;
719             best_score = score;
720         }
721     }
......
739 
740     pr_info("Machine model: %s\n", of_flat_dt_get_machine_name());
741 
742     return best_data;
743 }
```

第 714 行，通过函数 of_get_flat_dt_root 获取设备树根节点。

第 715~720 行，此循环就是查找匹配的 machine_desc 过程，第 716 行的 of_flat_dt_match 函数会将根节点 compatible 属性的值和每个 machine_desc 结构体中. dt_compat 的值进行比较，直至找到匹配的那个 machine_desc。

总结一下，Linux 内核通过根节点 compatible 属性找到对应的设备的函数调用过程，如图 43.3.4.2 所示：

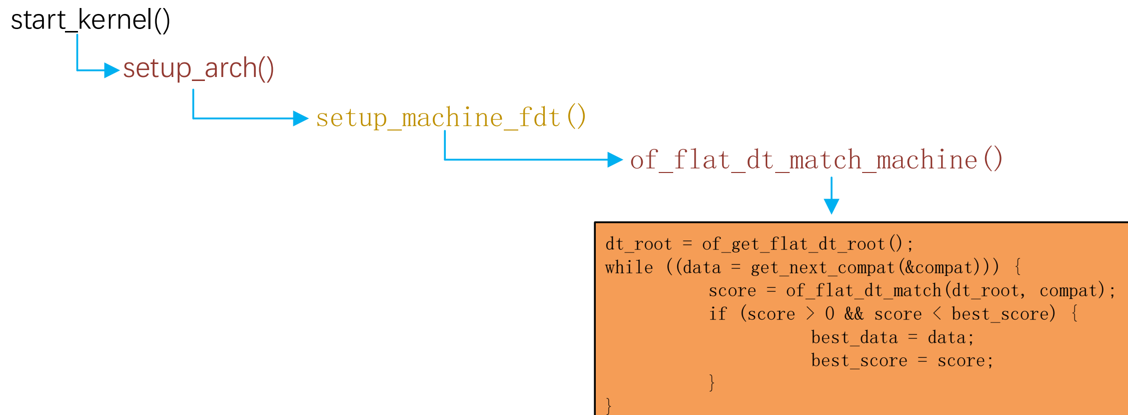


图 43.3.4.2 查找匹配设备的过程

43.3.5 向节点追加或修改内容

产品开发过程中可能面临着频繁的需求更改，比如第一版硬件上有一个 IIC 接口的六轴芯片 MPU6050，第二版硬件又要把这个 MPU6050 更换为 MPU9250 等。一旦硬件修改了，我们就要同步的修改设备树文件，毕竟设备树是描述板子硬件信息的文件。假设现在有个六轴芯片 fxls8471，fxls8471 要接到 I.MX6U-ALPHA 开发板的 I2C1 接口上，那么相当于需要在 i2c1 这个节点上添加一个 fxls8471 子节点。先看一下 I2C1 接口对应的节点，打开文件 imx6ull.dtsi 文件，找到如下所示内容：

示例代码 43.3.5.1 i2c1 节点

```
937 i2c1: i2c@021a0000 {
938     #address-cells = <1>;
939     #size-cells = <0>;
940     compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
941     reg = <0x021a0000 0x4000>;
942     interrupts = <GIC_SPI 36 IRQ_TYPE_LEVEL_HIGH>;
943     clocks = <&clks IMX6UL_CLK_I2C1>;
944     status = "disabled";
945 };
```

示例代码 43.3.5.1 就是 I.MX6ULL 的 I2C1 节点，现在要在 i2c1 节点下创建一个子节点，这个子节点就是 fxls8471，最简单的方法就是在 i2c1 下直接添加一个名为 fxls8471 的子节点，如下所示：

示例代码 43.3.5.2 添加 fxls8471 子节点

```
937 i2c1: i2c@021a0000 {
938     #address-cells = <1>;
939     #size-cells = <0>;
940     compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
941     reg = <0x021a0000 0x4000>;
942     interrupts = <GIC_SPI 36 IRQ_TYPE_LEVEL_HIGH>;
943     clocks = <&clks IMX6UL_CLK_I2C1>;
944     status = "disabled";
945
```

```
946     //fxls8471 子节点
947     fxls8471@1e {
948         compatible = "fsl,fxls8471";
949         reg = <0x1e>;
950     };
951 };
```

第 947~950 行就是添加的 fxls8471 这个芯片对应的子节点。但是这样会有个问题！i2c1 节点是定义在 imx6ull.dtsi 文件中的，而 imx6ull.dtsi 是设备树头文件，其他所有使用到 I.MX6ULL 这颗 SOC 的板子都会引用 imx6ull.dtsi 这个文件。直接在 i2c1 节点中添加 fxls8471 就相当于在其他的所有板子上都添加了 fxls8471 这个设备，但是其他的板子并没有这个设备啊！因此，按照示例代码 43.3.5.2 这样写肯定是不行的。

这里就要引入另外一个内容，那就是如何向节点追加数据，我们现在要解决的就是如何向 i2c1 节点追加一个名为 fxls8471 的子节点，而且不能影响到其他使用到 I.MX6ULL 的板子。I.MX6U-ALPHA 开发板使用的设备树文件为 imx6ull-alientek-emmc.dts，因此我们需要在 imx6ull-alientek-emmc.dts 文件中完成数据追加的内容，方式如下：

示例代码 43.3.5.3 节点追加数据方法

```
1 &i2c1 {
2   /* 要追加或修改的内容 */
3 };
```

第 1 行，&i2c1 表示要访问 i2c1 这个 label 所对应的节点，也就是 imx6ull.dtsi 中的“i2c1: i2c@021a0000”。

第 2 行，花括号内就是要向 i2c1 这个节点添加的内容，包括修改某些属性的值。

打开 imx6ull-alientek-emmc.dts，找到如下所示内容：

示例代码 43.3.5.4 向 i2c1 节点追加数据

```
224 &i2c1 {
225     clock-frequency = <100000>;
226     pinctrl-names = "default";
227     pinctrl-0 = <&pinctrl_i2c1>;
228     status = "okay";
229 
230     mag3110@0e {
231         compatible = "fsl,mag3110";
232         reg = <0x0e>;
233         position = <2>;
234     };
235 
236     fxls8471@1e {
237         compatible = "fsl,fxls8471";
238         reg = <0x1e>;
239         position = <0>;
240         interrupt-parent = <&gpio5>;
241         interrupts = <0 8>;
242     };
```

```
243 };
```

示例代码 43.3.5.4 就是向 i2c1 节点添加/修改数据，比如第 225 行的属性“clock-frequency”就表示 i2c1 时钟为 100KHz。“clock-frequency”就是新添加的属性。

第 228 行，将 status 属性的值由原来的 disabled 改为 okay。

第 230~234 行，i2c1 子节点 mag3110，因为 NXP 官方开发板在 I2C1 上接了一个磁力计芯片 mag3110，正点原子的 I.MX6U-ALPHA 开发板并没有使用 mag3110。

第 236~242 行，i2c1 子节点 fxls8471，同样是因为 NXP 官方开发板在 I2C1 上接了 fxls8471 这颗六轴芯片。

因为示例代码 43.3.5.4 中的内容是 imx6ull-alientek-emmc.dts 这个文件内的，所以不会对使用 I.MX6ULL 这颗 SOC 的其他板子造成任何影响。这个就是向节点追加或修改内容，重点就是通过&label 来访问节点，然后直接在里面编写要追加或者修改的内容。

43.4 创建小型模板设备树

上一节已经对 DTS 的语法做了比较详细的讲解，本节我们就根据前面讲解的语法，从头到尾编写一个小型的设备树文件。当然了，这个小型设备树没有实际的意义，做这个对的目的是为了掌握设备树的语法。在实际产品开发中，我们是不需要完完全全的重写一个.dts 设备树文件，一般都是使用 SOC 厂商提供好的.dts 文件，我们只需要在上面根据自己的实际情况做相应的修改即可。在编写设备树之前要先定义一个设备，我们就以 I.MX6ULL 这个 SOC 为例，我们需要在设备树里面描述的内容如下：

①、I.MX6ULL 这个 Cortex-A7 架构的 32 位 CPU。

②、I.MX6ULL 内部 ocram，起始地址 0x00900000，大小为 128KB(0x20000)。

③、I.MX6ULL 内部 aips1 域下的 ecspi1 外设控制器，寄存器起始地址为 0x02008000，大小为 0x4000。

④、I.MX6ULL 内部 aips2 域下的 usbotg1 外设控制器，寄存器起始地址为 0x02184000，大小为 0x4000。

⑤、I.MX6ULL 内部 aips3 域下的 rngb 外设控制器，寄存器起始地址为 0x02284000，大小为 0x4000。

为了简单起见，我们就在设备树里面就实现这些内容即可，首先，搭建一个仅含有根节点“/”的基础的框架，新建一个名为 myfirst.dts 文件，在里面输入如下所示内容：

示例代码 43.4.1 设备树基础框架

```
1 / {
2     compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3 }
```

设备树框架很简单，就一个根节点“/”，根节点里面只有一个 compatible 属性。我们就在这个基础框架上面将上面列出的内容一点点添加进来。

1、添加 cpus 节点

首先添加 CPU 节点，I.MX6ULL 采用 Cortex-A7 架构，而且只有一个 CPU，因此只有一个 cpu0 节点，完成以后如下所示：

示例代码 43.4.2 添加 CPU0 节点

```
1  / {
2     compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3
```

```
4       cpus {
5           #address-cells = <1>;
6           #size-cells = <0>;
7
8           //CPU0 节点
9           cpu0: cpu@0 {
10              compatible = "arm,cortex-a7";
11              device_type = "cpu";
12              reg = <0>;
13          };
14      };
15 }
```

第 4~14 行，cpus 节点，此节点用于描述 SOC 内部的所有 CPU，因为 I.MX6ULL 只有一个 CPU，因此只有一个 cpu0 子节点。

2、添加 soc 节点

像 uart，iic 控制器等等这些都属于 SOC 内部外设，因此一般会创建一个叫做 soc 的父节点来管理这些 SOC 内部外设的子节点，添加 soc 节点以后的 myfirst.dts 文件内容如下所示：

示例代码 43.4.3 添加 soc 节点

```
1  / {
2       compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3
4       cpus {
5           #address-cells = <1>;
6           #size-cells = <0>;
7
8           //CPU0 节点
9           cpu0: cpu@0 {
10              compatible = "arm,cortex-a7";
11              device_type = "cpu";
12              reg = <0>;
13          };
14      };
15
16      //soc 节点
17      soc {
18          #address-cells = <1>;
19          #size-cells = <1>;
20          compatible = "simple-bus";
21          ranges;
22      }
23 }
```

第 17~22 行，soc 节点，soc 节点设置#address-cells＝<1>，#size-cells＝<1>，这样 soc 子节点的 reg 属性中起始地占用一个字长，地址空间长度也占用一个字长。

第 21 行，ranges 属性，ranges 属性为空，说明子空间和父空间地址范围相同。

3、添加 ocram 节点

根据第②点的要求，添加 ocram 节点，ocram 是 I.MX6ULL 内部 RAM，因此 ocram 节点应该是 soc 节点的子节点。ocram 起始地址为 0x00900000，大小为 128KB(0x20000)，添加 ocram 节点以后 myfirst.dts 文件内容如下所示：

示例代码 43.4.4 添加 ocram 节点

```
1  / {
2      compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3
4      cpus {
5          #address-cells = <1>;
6          #size-cells = <0>;
7
8          //CPU0 节点
9          cpu0: cpu@0 {
10             compatible = "arm,cortex-a7";
11             device_type = "cpu";
12             reg = <0>;
13         };
14     };
15
16     //soc 节点
17     soc {
18         #address-cells = <1>;
19         #size-cells = <1>;
20         compatible = "simple-bus";
21         ranges;
22
23         //ocram 节点
24         ocram: sram@00900000 {
25             compatible = "fsl,lpm-sram";
26             reg = <0x00900000 0x20000>;
27         };
28     }
29 }
```

第 24~27 行，ocram 节点，第 24 行节点名字@后面的 0x00900000 就是 ocram 的起始地址。第 26 行的 reg 属性也指明了 ocram 内存的起始地址为 0x00900000，大小为 0x20000。

4、添加 aips1、aips2 和 aips3 这三个子节点

I.MX6ULL 内部分为三个域：aips1~3，这三个域分管不同的外设控制器，aips1~3 这三个域对应的内存范围如表 43.4.1 所示：

域	起始地址	大小(十六进制)
AIPS1	0X02000000	0X100000

AIPS2	0X02100000	0X100000
AIPS3	0X02200000	0X100000

表 43.4.1 aips1~3 地址范围

我们先在设备树中添加这三个域对应的子节点。aips1~3 这三个域都属于 soc 节点的子节点，完成以后的 myfirst.dts 文件内容如下所示：

示例代码 43.4.5 添加 aips1~3 节点

```
1  / {
2      compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3  
4      cpus {
5          #address-cells = <1>;
6          #size-cells = <0>;
7  
8          //CPU0 节点
9          cpu0: cpu@0 {
10             compatible = "arm,cortex-a7";
11             device_type = "cpu";
12             reg = <0>;
13         };
14     };
15 
16     //soc 节点
17     soc {
18         #address-cells = <1>;
19         #size-cells = <1>;
20         compatible = "simple-bus";
21         ranges;
22 
23         //ocram 节点
24         ocram: sram@00900000 {
25             compatible = "fsl,lpm-sram";
26             reg = <0x00900000 0x20000>;
27         };
28 
29         //aips1 节点
30         aips1: aips-bus@02000000 {
31             compatible = "fsl,aips-bus", "simple-bus";
32             #address-cells = <1>;
33             #size-cells = <1>;
34             reg = <0x02000000 0x100000>;
35             ranges;
36         }
37 
```

```
38          //aips2 节点
39          aips2: aips-bus@02100000 {
40              compatible = "fsl,aips-bus", "simple-bus";
41              #address-cells = <1>;
42              #size-cells = <1>;
43              reg = <0x02100000 0x100000>;
44              ranges;
45          }
46
47          //aips3 节点
48          aips3: aips-bus@02200000 {
49              compatible = "fsl,aips-bus", "simple-bus";
50              #address-cells = <1>;
51              #size-cells = <1>;
52              reg = <0x02200000 0x100000>;
53              ranges;
54          }
55      }
56 }
```

第 30~36 行，aips1 节点。

第 39~45 行，aips2 节点。

第 48~54 行，aips3 节点。

5、添加 ecspi1、usbotg1 和 rngb 这三个外设控制器节点

最后我们在 myfirst.dts 文件中加入 ecspi1，usbotg1 和 rngb 这三个外设控制器对应的节点，其中 ecspi1 属于 aips1 的子节点，usbotg1 属于 aips2 的子节点，rngb 属于 aips3 的子节点。最终的 myfirst.dts 文件内容如下：

示例代码 43.4.6 添加 ecspi1、usbotg1 和 rngb 这三个节点

```
1  / {
2      compatible = "fsl,imx6ull-alientek-evk", "fsl,imx6ull";
3
4      cpus {
5          #address-cells = <1>;
6          #size-cells = <0>;
7
8          //CPU0 节点
9          cpu0: cpu@0 {
10             compatible = "arm,cortex-a7";
11             device_type = "cpu";
12             reg = <0>;
13         };
14     };
15
16     //soc 节点
```

```
17      soc {
18          #address-cells = <1>;
19          #size-cells = <1>;
20          compatible = "simple-bus";
21          ranges;
22
23          //ocram 节点
24          ocram: sram@00900000 {
25              compatible = "fsl,lpm-sram";
26              reg = <0x00900000 0x20000>;
27          };
28
29          //aips1 节点
30          aips1: aips-bus@02000000 {
31              compatible = "fsl,aips-bus", "simple-bus";
32              #address-cells = <1>;
33              #size-cells = <1>;
34              reg = <0x02000000 0x100000>;
35              ranges;
36
37              //ecspi1 节点
38              ecspi1: ecspi@02008000 {
39                  #address-cells = <1>;
40                  #size-cells = <0>;
41                  compatible = "fsl,imx6ul-ecspi", "fsl,imx51-ecspi";
42                  reg = <0x02008000 0x4000>;
43                  status = "disabled";
44              };
45          }
46
47          //aips2 节点
48          aips2: aips-bus@02100000 {
49              compatible = "fsl,aips-bus", "simple-bus";
50              #address-cells = <1>;
51              #size-cells = <1>;
52              reg = <0x02100000 0x100000>;
53              ranges;
54
55              //usbotg1 节点
56              usbotg1: usb@02184000 {
57                  compatible = "fsl,imx6ul-usb", "fsl,imx27-usb";
58                  reg = <0x02184000 0x200>;
59                  status = "disabled";
```

```
60              };
61          }
62
63          //aips3 节点
64          aips3: aips-bus@02200000 {
65              compatible = "fsl,aips-bus", "simple-bus";
66              #address-cells = <1>;
67              #size-cells = <1>;
68              reg = <0x02200000 0x100000>;
69              ranges;
70
71              //rngb 节点
72              rngb: rngb@02284000 {
73                  compatible = "fsl,imx6sl-rng", "fsl,imx-rng", "imx-
rng";
74                  reg = <0x02284000 0x4000>;
75              };
76          }
77      }
78 }
```

第 38~44 行，ecspi1 外设控制器节点。

第 56~60 行，usbotg1 外设控制器节点。

第 72~75 行，rngb 外设控制器节点。

至此，myfirst.dts 这个小型的模板设备树就编写好了，基本和 imx6ull.dtsi 很像，可以看做是 imx6ull.dtsi 的缩小版。在 myfirst.dts 里面我们仅仅是编写了 I.MX6ULL 的外设控制器节点，像 IIC 接口，SPI 接口下所连接的具体设备我们并没有写，因为具体的设备其设备树属性内容不同，这个等到具体的实验在详细讲解。

43.5 设备树在系统中的体现

Linux 内核启动的时候会解析设备树中各个节点的信息，并且在根文件系统的/proc/device-tree 目录下根据节点名字创建不同文件夹，如图 43.5.1 所示：

```
/ # cd proc/device-tree/
/sys/firmware/devicetree/base # ls
#address-cells                  memory
#size-cells                     model
aliases                         name
backlight                       pxp_v4l2
chosen                          regulators
clocks                          reserved-memory
compatible                      soc
cpus                            sound
interrupt-controller@00a01000   spi4
/sys/firmware/devicetree/base #
```

根节点"/"的所有属性和子节点

图 43.5.1 根节点“/”的属性以及子节点

图 43.5.1 就是目录/proc/device-tree 目录下的内容，/proc/device-tree 目录下是根节点“/”的所有属性和子节点，我们依次来看一下这些属性和子节点。

1、根节点“/”各个属性

在图 43.5.1 中，根节点属性属性表现为一个个的文件(图中细字体文件)，比如图 43.5.1 中的“#address-cells”、“#size-cells”、“compatible”、“model”和“name”这 5 个文件，它们在设备树中就是根节点的 5 个属性。既然是文件那么肯定可以查看其内容，输入 cat 命令来查看 model 和 compatible 这两个文件的内容，结果如图 43.5.2 所示：

```
/sys/firmware/devicetree/base # cat model
Freescale i.MX6 ULL 14x14 EVK Board/sys/firmware/devicetree/base #
/sys/firmware/devicetree/base #
/sys/firmware/devicetree/base # cat compatible
fsl,imx6ull-14x14-evkfsl,imx6ull/sys/firmware/devicetree/base #
```
model文件内容
compatible文件内容

图 43.5.2 model 和 compatible 文件内容

从图 43.5.2 可以看出，文件 model 的内容是“Freescale i.MX6 ULL 14x14 EVK Board”，文件 compatible 的内容为“fsl,imx6ull-14x14-evkfsl,imx6ull”。打开文件 imx6ull-alientek-emmc.dts 查看一下，这不正是根节点“/”的 model 和 compatible 属性值吗！

2、根节点“/”各子节点

图 43.5.1 中各个文件夹(途中粗字体文件夹)就是根节点“/”的各个子节点，比如“aliases”、“backlight”、“chosen”和“clocks”等等。大家可以查看一下 imx6ull-alientek-emmc.dts 和 imx6ull.dtsi 这两个文件，看看根节点的子节点都有哪些，看看是否和图 43.5.1 中的一致。

/proc/device-tree 目录就是设备树在根文件系统中的体现，同样是按照树形结构组织的，进入/proc/device-tree/soc 目录中就可以看到 soc 节点的所有子节点，如图 43.5.3 所示：

```
/sys/firmware/devicetree/base # cd soc
/sys/firmware/devicetree/base/soc # ls
#address-cells         compatible             ranges
#size-cells            dma-apbh@01804000      sram@00900000
aips-bus@02000000      gpmi-nand@01806000     sram@00904000
aips-bus@02100000      interrupt-parent       sram@00905000
aips-bus@02200000      name
busfreq                pmu
/sys/firmware/devicetree/base/soc #
```

图 43.5.3 soc 节点的所有属性和子节点

和根节点“/”一样，图 43.5.3 中的所有文件分别为 soc 节点的属性文件和子节点文件夹。大家可以自行查看一下这些属性文件的内容是否和 imx6ull.dtsi 中 soc 节点的属性值相同，也可以进入“busfreq”这样的文件夹里面查看 soc 节点的子节点信息。

43.6 特殊节点

在根节点“/”中有两个特殊的子节点：aliases 和 chosen，我们接下来看一下这两个特殊的子节点。

43.6.1 aliases 子节点

打开 imx6ull.dtsi 文件，aliases 节点内容如下所示：

示例代码 43.6.1.1 aliases 子节点

```
18 aliases {
19     can0 = &flexcan1;
20     can1 = &flexcan2;
```

```
21      ethernet0 = &fec1;
22      ethernet1 = &fec2;
23      gpio0 = &gpio1;
24      gpio1 = &gpio2;
......
42      spi0 = &ecspi1;
43      spi1 = &ecspi2;
44      spi2 = &ecspi3;
45      spi3 = &ecspi4;
46      usbphy0 = &usbphy1;
47      usbphy1 = &usbphy2;
48  };
```

单词 aliases 的意思是“别名”，因此 aliases 节点的主要功能就是定义别名，定义别名的目的就是为了方便访问节点。不过我们一般会在节点命名的时候会加上 label，然后通过&label 来访问节点，这样也很方便，而且设备树里面大量的使用&label 的形式来访问节点。

43.6.2 chosen 子节点

chosen 并不是一个真实的设备，chosen 节点主要是为了 uboot 向 Linux 内核传递数据，重点是 bootargs 参数。一般.dts 文件中 chosen 节点通常为空或者内容很少，imx6ull-alientek-emmc.dts 中 chosen 节点内容如下所示：

示例代码 43.6.2.1 chosen 子节点

```
18  chosen {
19      stdout-path = &uart1;
20  };
```

从示例代码 43.6.2.1 中可以看出，chosen 节点仅仅设置了属性“stdout-path”，表示标准输出使用 uart1。但是当我们进入到/proc/device-tree/chosen 目录里面，会发现多了 bootargs 这个属性，如图 43.6.2.1 所示：

```
/sys/firmware/devicetree/base/chosen # ls
bootargs          name          stdout-path
/sys/firmware/devicetree/base/chosen #
```

图 43.6.2.1 chosen 节点目录

输入 cat 命令查看 bootargs 这个文件的内容，结果如图 43.6.2.2 所示：

```
/sys/firmware/devicetree/base/chosen # cat bootargs
console=ttymxc0,115200 root=/dev/nfs rw nfsroot=192.168.1.250:/home/zuozhongkai/
linux/nfs/rootfs ip=192.168.1.251:192.168.1.250:192.168.1.1:255.255.255.0::eth0:
off/sys/firmware/devicetree/base/chosen #
```

图 43.6.2.2 bootargs 文件内容

从图 43.6.2.2 可以看出，bootargs 这个文件的内容为“console=ttymxc0,115200……”，这个不就是我们在 uboot 中设置的 bootargs 环境变量的值吗？现在有两个疑点：

①、我们并没有在设备树中设置 chosen 节点的 bootargs 属性，那么图 43.6.2.1 中 bootargs 这个属性是怎么产生的？

②、为何 bootargs 文件的内容和 uboot 中 bootargs 环境变量的值一样？它们之间有什么关系？

前面讲解 uboot 的时候说过，uboot 在启动 Linux 内核的时候会将 bootargs 的值传递给 Linux 内核，bootargs 会作为 Linux 内核的命令行参数，Linux 内核启动的时候会打印出命令行参数(也就是 uboot 传递进来的 bootargs 的值)，如图 43.6.2.3 所示：

```
Booting Linux on physical CPU 0x0
Linux version 4.1.15 (zuozhongkai@ubuntu) (gcc version 4.9.4 (Linaro GCC 4.9-201
7.01) ) #2 SMP PREEMPT Thu Jun 27 20:43:04 CST 2019
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=10c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Freescale i.MX6 ULL 14x14 EVK Board
Reserved memory: created CMA memory pool at 0x8c000000, size 320 MiB
Reserved memory: initialized node linux,cma, compatible id shared-dma-pool
Memory policy: Data cache writealloc
PERCPU: Embedded 12 pages/cpu @8bb32000 s16768 r8192 d24192 u49152
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttymxc0,115200 root=/dev/nfs rw nfsroot=192.168.1.2
50:/home/zuozhongkai/linux/nfs/rootfs ip=192.168.1.251:192.168.1.250:192.168.1.1
:255.255.255.0::eth0:off
```

命令行参数

图 43.6.2.3 命令行参数

既然 chosen 节点的 bootargs 属性不是我们在设备树里面设置的，那么只有一种可能，那就是 uboot 自己在 chosen 节点里面添加了 bootargs 属性！并且设置 bootargs 属性的值为 bootargs 环境变量的值。因为在启动 Linux 内核之前，只有 uboot 知道 bootargs 环境变量的值，并且 uboot 也知道.dtb 设备树文件在 DRAM 中的位置，因此 uboot 的“作案”嫌疑最大。在 uboot 源码中全局搜索“chosen”这个字符串，看看能不能找到一些蛛丝马迹。果然不出所料，在 common/fdt_support.c 文件中发现了“chosen”的身影，fdt_support.c 文件中有个 fdt_chosen 函数，此函数内容如下所示：

示例代码 43.6.2.2 uboot 源码中的 fdt_chosen 函数

```
275 int fdt_chosen(void *fdt)
276 {
277     int   nodeoffset;
278     int   err;
279     char  *str;     /* used to set string properties */
280 
281     err = fdt_check_header(fdt);
282     if (err < 0) {
283         printf("fdt_chosen: %s\n", fdt_strerror(err));
284         return err;
285     }
286 
287     /* find or create "/chosen" node. */
288     nodeoffset = fdt_find_or_add_subnode(fdt, 0, "chosen");
289     if (nodeoffset < 0)
290         return nodeoffset;
291 
292     str = getenv("bootargs");
293     if (str) {
294         err = fdt_setprop(fdt, nodeoffset, "bootargs", str,
295                  strlen(str) + 1);
296         if (err < 0) {
```

```
297             printf("WARNING: could not set bootargs %s.\n",
298                    fdt_strerror(err));
299             return err;
300         }
301     }
302 
303     return fdt_fixup_stdout(fdt, nodeoffset);
304 }
```

第 288 行，调用函数 fdt_find_or_add_subnode 从设备树(.dtb)中找到 chosen 节点，如果没有找到的话就会自己创建一个 chosen 节点。

第 292 行，读取 uboot 中 bootargs 环境变量的内容。

第 294 行，调用函数 fdt_setprop 向 chosen 节点添加 bootargs 属性，并且 bootargs 属性的值就是环境变量 bootargs 的内容。

证据“石锤”了，就是 uboot 中的 fdt_chosen 函数在设备树的 chosen 节点中加入了 bootargs 属性，并且还设置了 bootargs 属性值。接下来我们顺着 fdt_chosen 函数一点点的抽丝剥茧，看看都有哪些函数调用了 fdt_chosen，一直找到最终的源头。这里我就不卖关子了，直接告诉大家整个流程是怎么样的，见图 43.6.2.4：

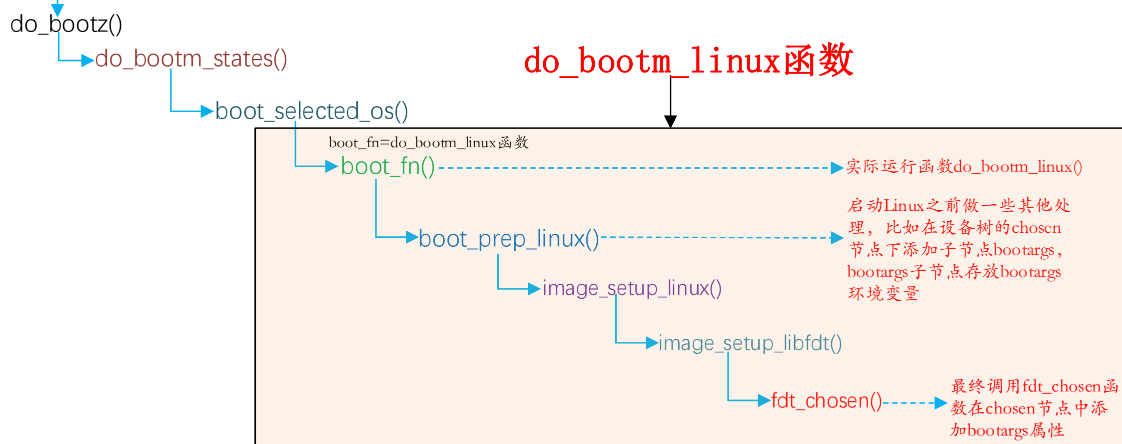


图 43.6.2.4 fdt_chosen 函数调用流程

图 43.6.2.4 中框起来的部分就是函数 do_bootm_linux 函数的执行流程，也就是说 do_bootm_linux 函数会通过一系列复杂的调用，最终通过 fdt_chosen 函数在 chosen 节点中加入了 bootargs 属性。而我们通过 bootz 命令启动 Linux 内核的时候会运行 do_bootm_linux 函数，至此，真相大白，一切事情的源头都源于如下命令：

```
bootz 80800000 – 83000000
```

当我们输入上述命令并执行以后，do_bootz 函数就会执行，然后一切就按照图 43.6.2.4 中所示的流程开始运行。

43.7 Linux 内核解析 DTB 文件

Linux 内核在启动的时候会解析 DTB 文件，然后在/proc/device-tree 目录下生成相应的设备树节点文件。接下来我们简单分析一下 Linux 内核是如何解析 DTB 文件的，流程如图 43.7.1 所示：

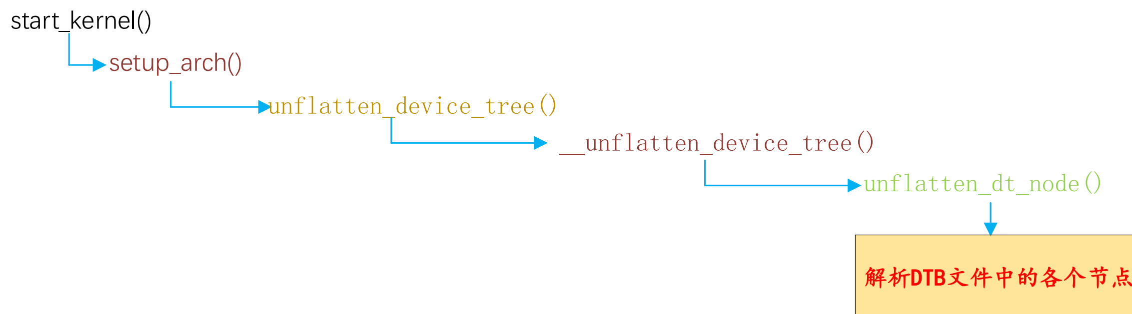


图 43.7.1 设备树节点解析流程。

从图 43.7.1 中可以看出，在 start_kernel 函数中完成了设备树节点解析的工作，最终实际工作的函数为 unflatten_dt_node。

43.8 绑定信息文档

设备树是用来描述板子上的设备信息的，不同的设备其信息不同，反映到设备树中就是属性不同。那么我们在设备树中添加一个硬件对应的节点的时候从哪里查阅相关的说明呢？在 Linux 内核源码中有详细的.txt 文档描述了如何添加节点，这些.txt 文档叫做绑定文档，路径为：Linux 源码目录/Documentation/devicetree/bindings，如图 43.8.1 所示：

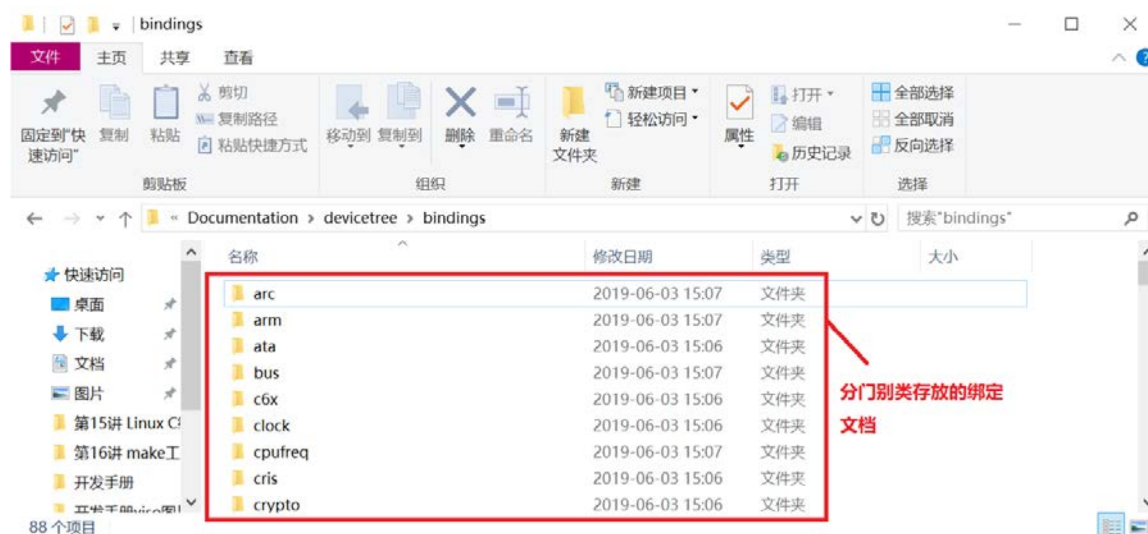


图 43.8.1 绑定文档

比如我们现在要想在 I.MX6ULL 这颗 SOC 的 I2C 下添加一个节点，那么就可以查看 Documentation/devicetree/bindings/i2c/i2c-imx.txt，此文档详细的描述了 I.MX 系列的 SOC 如何在设备树中添加 I2C 设备节点，文档内容如下所示：

```
* Freescale Inter IC (I2C) and High Speed Inter IC (HS-I2C) for i.MX

Required properties:
- compatible :
  - "fsl,imx1-i2c" for I2C compatible with the one integrated on i.MX1
SoC
  - "fsl,imx21-i2c" for I2C compatible with the one integrated on i.MX21
SoC
```

```
  - "fsl,vf610-i2c" for I2C compatible with the one integrated on Vybrid
vf610 SoC
- reg : Should contain I2C/HS-I2C registers location and length
- interrupts : Should contain I2C/HS-I2C interrupt
- clocks : Should contain the I2C/HS-I2C clock specifier

Optional properties:
- clock-frequency : Constains desired I2C/HS-I2C bus clock frequency in
Hz.
  The absence of the propoerty indicates the default frequency 100 kHz.
- dmas: A list of two dma specifiers, one for each entry in dma-names.
- dma-names: should contain "tx" and "rx".

Examples:

i2c@83fc4000 { /* I2C2 on i.MX51 */
    compatible = "fsl,imx51-i2c", "fsl,imx21-i2c";
    reg = <0x83fc4000 0x4000>;
    interrupts = <63>;
};

i2c@70038000 { /* HS-I2C on i.MX51 */
    compatible = "fsl,imx51-i2c", "fsl,imx21-i2c";
    reg = <0x70038000 0x4000>;
    interrupts = <64>;
    clock-frequency = <400000>;
};

i2c0: i2c@40066000 { /* i2c0 on vf610 */
    compatible = "fsl,vf610-i2c";
    reg = <0x40066000 0x1000>;
    interrupts =<0 71 0x04>;
    dmas = <&edma0 0 50>,
        <&edma0 0 51>;
    dma-names = "rx","tx";
};
```

有时候使用的一些芯片在 Documentation/devicetree/bindings 目录下找不到对应的文档，这个时候就要咨询芯片的提供商，让他们给你提供参考的设备树文件。

43.9 设备树常用 OF 操作函数

设备树描述了设备的详细信息，这些信息包括数字类型的、字符串类型的、数组类型的，我们在编写驱动的时候需要获取到这些信息。比如设备树使用 reg 属性描述了某个外设的寄存器地址为 0X02005482，长度为 0X400，我们在编写驱动的时候需要获取到 reg 属性的

0X02005482 和 0X400 这两个值，然后初始化外设。Linux 内核给我们提供了一系列的函数来获取设备树中的节点或者属性信息，这一系列的函数都有一个统一的前缀“of_”，所以在很多资料里面也被叫做 OF 函数。这些 OF 函数原型都定义在 include/linux/of.h 文件中。

43.9.1 查找节点的 OF 函数

设备都是以节点的形式“挂”到设备树上的，因此要想获取这个设备的其他属性信息，必须先获取到这个设备的节点。Linux 内核使用 device_node 结构体来描述一个节点，此结构体定义在文件 include/linux/of.h 中，定义如下：

示例代码 43.3.9.1 device_node 节点

```
49 struct device_node {
50     const char *name;              /* 节点名字         */
51     const char *type;              /* 设备类型         */
52     phandle phandle;
53     const char *full_name;         /* 节点全名         */
54     struct fwnode_handle fwnode;
55 
56     struct  property *properties;  /* 属性             */
57     struct  property *deadprops;   /* removed 属性     */
58     struct  device_node *parent;   /* 父节点           */
59     struct  device_node *child;    /* 子节点           */
60     struct  device_node *sibling;
61     struct  kobject kobj;
62     unsigned long _flags;
63     void    *data;
64 #if defined(CONFIG_SPARC)
65     const char *path_component_name;
66     unsigned int unique_id;
67     struct of_irq_controller *irq_trans;
68 #endif
69 };
```

与查找节点有关的 OF 函数有 5 个，我们依次来看一下。

1、of_find_node_by_name 函数

of_find_node_by_name 函数通过节点名字查找指定的节点，函数原型如下：

```
struct device_node *of_find_node_by_name(struct device_node    *from,
                                         const char            *name);
```

函数参数和返回值含义如下：

from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

name：要查找的节点名字。

返回值：找到的节点，如果为 NULL 表示查找失败。

2、of_find_node_by_type 函数

of_find_node_by_type 函数通过 device_type 属性查找指定的节点，函数原型如下：

```
struct device_node *of_find_node_by_type(struct device_node *from, const char *type)
```

函数参数和返回值含义如下：

from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

type：要查找的节点对应的 type 字符串，也就是 device_type 属性值。

返回值：找到的节点，如果为 NULL 表示查找失败。

3、of_find_compatible_node 函数

of_find_compatible_node 函数根据 device_type 和 compatible 这两个属性查找指定的节点，函数原型如下：

```
struct device_node *of_find_compatible_node(struct device_node    *from,
                                            const char            *type,
                                            const char            *compatible)
```

函数参数和返回值含义如下：

from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

type：要查找的节点对应的 type 字符串，也就是 device_type 属性值，可以为 NULL，表示忽略掉 device_type 属性。

compatible：要查找的节点所对应的 compatible 属性列表。

返回值：找到的节点，如果为 NULL 表示查找失败

4、of_find_matching_node_and_match 函数

of_find_matching_node_and_match 函数通过 of_device_id 匹配表来查找指定的节点，函数原型如下：

```
struct device_node *of_find_matching_node_and_match(struct device_node        *from,
                                                    const struct of_device_id *matches,
                                                    const struct of_device_id **match)
```

函数参数和返回值含义如下：

from：开始查找的节点，如果为 NULL 表示从根节点开始查找整个设备树。

matches：of_device_id 匹配表，也就是在此匹配表里面查找节点。

match：找到的匹配的 of_device_id。

返回值：找到的节点，如果为 NULL 表示查找失败

5、of_find_node_by_path 函数

of_find_node_by_path 函数通过路径来查找指定的节点，函数原型如下：

```
inline struct device_node *of_find_node_by_path(const char *path)
```

函数参数和返回值含义如下：

path：带有全路径的节点名，可以使用节点的别名，比如“/backlight”就是 backlight 这个节点的全路径。

返回值：找到的节点，如果为 NULL 表示查找失败

43.9.2 查找父/子节点的 OF 函数

Linux 内核提供了几个查找节点对应的父节点或子节点的 OF 函数，我们依次来看一下。

1、of_get_parent 函数

of_get_parent 函数用于获取指定节点的父节点(如果有父节点的话)，函数原型如下：

```
struct device_node *of_get_parent(const struct device_node *node)
```

函数参数和返回值含义如下：

node：要查找的父节点的节点。

返回值：找到的父节点。

2、of_get_next_child 函数

of_get_next_child 函数用迭代的查找子节点，函数原型如下：

```
struct device_node *of_get_next_child(const struct device_node      *node,
                                      struct device_node            *prev)
```

函数参数和返回值含义如下：

node：父节点。

prev：前一个子节点，也就是从哪一个子节点开始迭代的查找下一个子节点。可以设置为 NULL，表示从第一个子节点开始。

返回值：找到的下一个子节点。

43.9.3 提取属性值的 OF 函数

节点的属性信息里面保存了驱动所需要的内容，因此对于属性值的提取非常重要，Linux 内核中使用结构体 property 表示属性，此结构体同样定义在文件 include/linux/of.h 中，内容如下：

示例代码 43.9.3.1 property 结构体

```
35 struct property {
36     char    *name;             /* 属性名字     */
37     int length;                /* 属性长度     */
38     void    *value;            /* 属性值       */
39     struct property *next;     /* 下一个属性   */
40     unsigned long _flags;
41     unsigned int unique_id;
42     struct bin_attribute attr;
43 };
```

Linux 内核也提供了提取属性值的 OF 函数，我们依次来看一下。

1、of_find_property 函数

of_find_property 函数用于查找指定的属性，函数原型如下：

```
property *of_find_property(const struct device_node   *np,
                           const char                 *name,
                           int                        *lenp)
```

函数参数和返回值含义如下：

np：设备节点。

name：　属性名字。

lenp：属性值的字节数

返回值：找到的属性。

2、of_property_count_elems_of_size 函数

of_property_count_elems_of_size 函数用于获取属性中元素的数量，比如 reg 属性值是一个数组，那么使用此函数可以获取到这个数组的大小，此函数原型如下：

```
int of_property_count_elems_of_size(const struct device_node  *np,
                                    const char                *propname,
```

```
int elem_size)
```

函数参数和返回值含义如下：

np：设备节点。

proname： 需要统计元素数量的属性名字。

elem_size：元素长度。

返回值：得到的属性元素数量。

3、of_property_read_u32_index 函数

of_property_read_u32_index 函数用于从属性中获取指定标号的 u32 类型数据值(无符号 32 位)，比如某个属性有多个 u32 类型的值，那么就可以使用此函数来获取指定标号的数据值，此函数原型如下：

```
int of_property_read_u32_index(const struct device_node *np,
                               const char               *propname,
                               u32                      index,
                               u32                      *out_value)
```

函数参数和返回值含义如下：

np：设备节点。

proname： 要读取的属性名字。

index：要读取的值标号。

out_value：读取到的值

返回值：0 读取成功，负值，读取失败，-EINVAL 表示属性不存在，-ENODATA 表示没有要读取的数据，-EOVERFLOW 表示属性值列表太小。

4、 of_property_read_u8_array 函数
of_property_read_u16_array 函数
of_property_read_u32_array 函数
of_property_read_u64_array 函数

这 4 个函数分别是读取属性中 u8、u16、u32 和 u64 类型的数组数据，比如大多数的 reg 属性都是数组数据，可以使用这 4 个函数一次读取出 reg 属性中的所有数据。这四个函数的原型如下：

```
int of_property_read_u8_array(const struct device_node *np,
                              const char               *propname,
                              u8                       *out_values,
                              size_t                   sz)
int of_property_read_u16_array(const struct device_node *np,
                               const char               *propname,
                               u16                      *out_values,
                               size_t                   sz)
int of_property_read_u32_array(const struct device_node *np,
                               const char               *propname,
                               u32                      *out_values,
                               size_t                   sz)
int of_property_read_u64_array(const struct device_node *np,
                               const char               *propname,
                               u64                      *out_values,
```

```
                    size_t                    sz)
```

函数参数和返回值含义如下：

np：设备节点。

proname： 要读取的属性名字。

out_value：读取到的数组值，分别为 u8、u16、u32 和 u64。

sz：要读取的数组元素数量。

返回值：0，读取成功，负值，读取失败，-EINVAL 表示属性不存在，-ENODATA 表示没有要读取的数据，-EOVERFLOW 表示属性值列表太小。

5、of_property_read_u8 函数

of_property_read_u16 函数

of_property_read_u32 函数

of_property_read_u64 函数

有些属性只有一个整形值，这四个函数就是用于读取这种只有一个整形值的属性，分别用于读取 u8、u16、u32 和 u64 类型属性值，函数原型如下：

```
int of_property_read_u8(const struct device_node     *np,
                        const char                   *propname,
                        u8                           *out_value)
int of_property_read_u16(const struct device_node    *np,
                        const char                   *propname,
                        u16                          *out_value)
int of_property_read_u32(const struct device_node    *np,
                        const char                   *propname,
                        u32                          *out_value)
int of_property_read_u64(const struct device_node    *np,
                        const char                   *propname,
                        u64                          *out_value)
```

函数参数和返回值含义如下：

np：设备节点。

proname： 要读取的属性名字。

out_value：读取到的数组值。

返回值：0，读取成功，负值，读取失败，-EINVAL 表示属性不存在，-ENODATA 表示没有要读取的数据，-EOVERFLOW 表示属性值列表太小。

6、of_property_read_string 函数

of_property_read_string 函数用于读取属性中字符串值，函数原型如下：

```
int of_property_read_string(struct device_node   *np,
                            const char           *propname,
                            const char           **out_string)
```

函数参数和返回值含义如下：

np：设备节点。

proname： 要读取的属性名字。

out_string：读取到的字符串值。

返回值：0，读取成功，负值，读取失败。

7、of_n_addr_cells 函数

of_n_addr_cells 函数用于获取#address-cells 属性值，函数原型如下：

```
int of_n_addr_cells(struct device_node *np)
```

函数参数和返回值含义如下：

np：设备节点。

返回值：获取到的#address-cells 属性值。

8、of_n_size_cells 函数

of_size_cells 函数用于获取#size-cells 属性值，函数原型如下：

```
int of_n_size_cells(struct device_node *np)
```

函数参数和返回值含义如下：

np：设备节点。

返回值：获取到的#size-cells 属性值。

43.9.4 其他常用的 OF 函数

1、of_device_is_compatible 函数

of_device_is_compatible 函数用于查看节点的 compatible 属性是否有包含 compat 指定的字符串，也就是检查设备节点的兼容性，函数原型如下：

```
int of_device_is_compatible(const struct device_node     *device,
                            const char                   *compat)
```

函数参数和返回值含义如下：

device：设备节点。

compat：要查看的字符串。

返回值：0，节点的 compatible 属性中包含 compat 指定的字符串；其他值，节点的 compatible 属性中不包含 compat 指定的字符串。

2、of_get_address 函数

of_get_address 函数用于获取地址相关属性，主要是“reg”或者“assigned-addresses”属性值，函数属性如下：

```
const __be32 *of_get_address(struct device_node     *dev,
                             int                    index,
                             u64                    *size,
                             unsigned int           *flags)
```

函数参数和返回值含义如下：

dev：设备节点。

index：要读取的地址标号。

size：地址长度。

flags：参数，比如 IORESOURCE_IO、IORESOURCE_MEM 等

返回值：读取到的地址数据首地址，为 NULL 的话表示读取失败。

3、of_translate_address 函数

of_translate_address 函数负责将从设备树读取到的地址转换为物理地址，函数原型如下：

```
u64 of_translate_address(struct device_node     *dev,
```

```
const __be32 *in_addr)
```

函数参数和返回值含义如下：

dev：设备节点。

in_addr：要转换的地址。

返回值：得到的物理地址，如果为 OF_BAD_ADDR 的话表示转换失败。

4、of_address_to_resource 函数

IIC、SPI、GPIO 等这些外设都有对应的寄存器，这些寄存器其实就是一组内存空间，Linux 内核使用 resource 结构体来描述一段内存空间，"resource"翻译出来就是"资源"，因此用 resource 结构体描述的都是设备资源信息，resource 结构体定义在文件 include/linux/ioport.h 中，定义如下：

示例代码 43.9.4.1 resource 结构体

```
18 struct resource {
19     resource_size_t start;
20     resource_size_t end;
21     const char *name;
22     unsigned long flags;
23     struct resource *parent, *sibling, *child;
24 };
```

对于 32 位的 SOC 来说，resource_size_t 是 u32 类型的。其中 start 表示开始地址，end 表示结束地址，name 是这个资源的名字，flags 是资源标志位，一般表示资源类型，可选的资源标志定义在文件 include/linux/ioport.h 中，如下所示：

示例代码 43.9.4.2 资源标志

```
1  #define IORESOURCE_BITS          0x000000ff
2  #define IORESOURCE_TYPE_BITS     0x00001f00
3  #define IORESOURCE_IO            0x00000100
4  #define IORESOURCE_MEM           0x00000200
5  #define IORESOURCE_REG           0x00000300
6  #define IORESOURCE_IRQ           0x00000400
7  #define IORESOURCE_DMA           0x00000800
8  #define IORESOURCE_BUS           0x00001000
9  #define IORESOURCE_PREFETCH      0x00002000
10 #define IORESOURCE_READONLY      0x00004000
11 #define IORESOURCE_CACHEABLE     0x00008000
12 #define IORESOURCE_RANGELENGTH   0x00010000
13 #define IORESOURCE_SHADOWABLE    0x00020000
14 #define IORESOURCE_SIZEALIGN     0x00040000
15 #define IORESOURCE_STARTALIGN    0x00080000
16 #define IORESOURCE_MEM_64        0x00100000
17 #define IORESOURCE_WINDOW        0x00200000
18 #define IORESOURCE_MUXED         0x00400000
19 #define IORESOURCE_EXCLUSIVE     0x08000000
20 #define IORESOURCE_DISABLED      0x10000000
21 #define IORESOURCE_UNSET         0x20000000
```

```
22 #define IORESOURCE_AUTO          0x40000000
23 #define IORESOURCE_BUSY          0x80000000
```

大家一般最常见的资源标志就是 IORESOURCE_MEM、IORESOURCE_REG 和 IORESOURCE_IRQ 等。接下来我们回到 of_address_to_resource 函数，此函数看名字像是从设备树里面提取资源值，但是本质上就是将 reg 属性值，然后将其转换为 resource 结构体类型，函数原型如下所示

```
int of_address_to_resource(struct device_node    *dev,
                           int                   index,
                           struct resource       *r)
```

函数参数和返回值含义如下：

dev：设备节点。

index：地址资源标号。

r：得到的 resource 类型的资源值。

返回值：0，成功；负值，失败。

5、of_iomap 函数

of_iomap 函数用于直接内存映射，以前我们会通过 ioremap 函数来完成物理地址到虚拟地址的映射，采用设备树以后就可以直接通过 of_iomap 函数来获取内存地址所对应的虚拟地址，不需要使用 ioremap 函数了。当然了，你也可以使用 ioremap 函数来完成物理地址到虚拟地址的内存映射，只是在采用设备树以后，大部分的驱动都使用 of_iomap 函数了。of_iomap 函数本质上也是将 reg 属性中地址信息转换为虚拟地址，如果 reg 属性有多段的话，可以通过 index 参数指定要完成内存映射的是那一段，of_iomap 函数原型如下：

```
void __iomem *of_iomap(struct device_node    *np,
                       int                   index)
```

函数参数和返回值含义如下：

np：设备节点。

index：reg 属性中要完成内存映射的段，如果 reg 属性只有一段的话 index 就设置为 0。

返回值：经过内存映射后的虚拟内存首地址，如果为 NULL 的话表示内存映射失败。

关于设备树常用的 OF 函数就先讲解到这里，Linux 内核中关于设备树的 OF 函数不仅仅只有前面讲的这几个，还有很多 OF 函数我们并没有讲解，这些没有讲解的 OF 函数要结合具体的驱动，比如获取中断号的 OF 函数、获取 GPIO 的 OF 函数等等，这些 OF 函数我们在后面的驱动实验中再详细的讲解。

关于设备树就讲解到这里，关于设备树我们重点要了解一下几点内容：

①、DTS、DTB 和 DTC 之间的区别，如何将.dts 文件编译为.dtb 文件。

②、设备树语法，这个是重点，因为在实际工作中我们是需要修改设备树的。

③、设备树的几个特殊子节点。

④、关于设备树的 OF 操作函数，也是重点，因为设备树最终是被驱动文件所使用的，而驱动文件必须要读取设备树中的属性信息，比如内存信息、GPIO 信息、中断信息等等。要想在驱动中读取设备树的属性值，那么就必须使用 Linux 内核提供的众多的 OF 函数。

从下一章开始所以的 Linux 驱动实验都将采用设备树，从最基本的点灯，到复杂的音频、网络或块设备等驱动。将会带领大家由简入深，深度剖析设备树，最终掌握基于设备树的驱动开发技能。

第四十四章 设备树下的 LED 驱动实验

上一章我们详细的讲解了设备树语法以及在驱动开发中常用的 OF 函数，本章我们就开始第一个基于设备树的 Linux 驱动实验。本章在第四十二章实验的基础上完成，只是将其驱动开发改为设备树形式而已。

44.1 设备树 LED 驱动原理

在《第四十二章 新字符设备驱动实验》中，我们直接在驱动文件 newchrled.c 中定义有关寄存器物理地址，然后使用 io_remap 函数进行内存映射，得到对应的虚拟地址，最后操作寄存器对应的虚拟地址完成对 GPIO 的初始化。本章我们在第四十二章实验基础上完成，本章我们使用设备树来向 Linux 内核传递相关的寄存器物理地址，Linux 驱动文件使用上一章讲解的 OF 函数从设备树中获取所需的属性值，然后使用获取到的属性值来初始化相关的 IO。本章实验还是比较简单的，本章实验重点内容如下：

①、在 imx6ull-alientek-emmc.dts 文件中创建相应的设备节点。

②、编写驱动程序(在第四十二章实验基础上完成)，获取设备树中的相关属性值。

③、使用获取到的有关属性值来初始化 LED 所使用的 GPIO。

44.2 硬件原理图分析

本章实验硬件原理图参考 8.3 小节即可。

44.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 4_dtsled。

本章实验在四十二章实验的基础上完成，重点是将驱动改为基于设备树的.

44.3.1 修改设备树文件

在根节“/”下创建一个名为“alphaled”的子节点，打开 imx6ull-alientek-emmc.dts 文件，在根节点“/”最后面输入如下所示内容：

示例代码 44.3.1.1 alphaled 节点

```
1  alphaled {
2      #address-cells = <1>;
3      #size-cells = <1>;
4      compatible = "atkalpha-led";
5      status = "okay";
6      reg = <   0X020C406C 0X04      /* CCM_CCGR1_BASE          */
7                0X020E0068 0X04      /* SW_MUX_GPIO1_IO03_BASE  */
8                0X020E02F4 0X04      /* SW_PAD_GPIO1_IO03_BASE  */
9                0X0209C000 0X04      /* GPIO1_DR_BASE           */
10               0X0209C004 0X04 >;   /* GPIO1_GDIR_BASE         */
11 };
```

第 2、3 行，属性#address-cells 和#size-cells 都为 1，表示 reg 属性中起始地址占用一个字长(cell)，地址长度也占用一个字长(cell)。

第 4 行，属性 compatbile 设置 alphaled 节点兼容性为“atkalpha-led”。

第 5 行，属性 status 设置状态为“okay”。

第 6~10 行，reg 属性，非常重要！reg 属性设置了驱动里面所要使用的寄存器物理地址，比如第 6 行的“0X020C406C 0X04”表示 I.MX6ULL 的 CCM_CCGR1 寄存器，其中寄存器首地址为 0X020C406C，长度为 4 个字节。

设备树修改完成以后输入如下命令重新编译一下 imx6ull-alientek-emmc.dts：

```
make dtbs
```

编译完成以后得到 imx6ull-alientek-emmc.dtb，使用新的 imx6ull-alientek-emmc.dtb 启动 Linux 内核。Linux 启动成功以后进入到/proc/device-tree/目录中查看是否有“alphaled”这个节点，结果如图 44.3.1.1 所示：

```
/ # cd /proc/device-tree/
/sys/firmware/devicetree/base # ls
#address-cells                  memory
#size-cells                     model
aliases                         name
alphaled                        pxp_v4l2
backlight                       regulators
chosen                          reserved-memory
clocks                          soc
compatible                      sound
cpus                            spi4
interrupt-controller@00a01000
```

alphaled节点

图 44.3.1.1 alphaled 节点

如果没有“alphaled”节点的话请重点下面两点：

①、检查设备树修改是否成功，也就是 alphaled 节点是否为根节点“/”的子节点。

②、检查是否使用新的设备树启动的 Linux 内核。

可以进入到图 44.3.1 中的 alphaled 目录中，查看一下都有哪些属性文件，结果如图 44.3.1.2 所示：

```
/sys/firmware/devicetree/base/alphaled # ls
#address-cells  compatible      reg
#size-cells     name            status
/sys/firmware/devicetree/base/alphaled #
```

图 44.3.1.2 alphaled 节点文件

大家可以查看一下 compatible、status 等属性值是否和我们设置的一致。

44.3.2 LED 灯驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第四十二章实验驱动文件 newchrled.c 的基础上修改而来。新建名为“4_dtsled”文件夹，然后在 4_dtsled 文件夹里面创建 vscode 工程，工作区命名为“dtsled”。工程创建好以后新建 dtsled.c 文件，在 dtsled.c 里面输入如下内容：

示例代码 44.3.2.1 dtsled.c 文件内容

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of.h>
12  #include <linux/of_address.h>
```

```
13 #include <asm/mach/map.h>
14 #include <asm/uaccess.h>
15 #include <asm/io.h>
16 /***************************************************************
17 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
18 文件名      : dtsled.c
19 作者        : 左忠凯
20 版本        : V1.0
21 描述        : LED 驱动文件。
22 其他        : 无
23 论坛        : www.openedv.com
24 日志        : 初版 V1.0 2019/7/9 左忠凯创建
25 ***************************************************************/
26 #define DTSLED_CNT       1          /* 设备号个数  */
27 #define DTSLED_NAME      "dtsled"   /* 名字        */
28 #define LEDOFF           0          /* 关灯        */
29 #define LEDON            1          /* 开灯        */
30 
31 /* 映射后的寄存器虚拟地址指针 */
32 static void __iomem *IMX6U_CCM_CCGR1;
33 static void __iomem *SW_MUX_GPIO1_IO03;
34 static void __iomem *SW_PAD_GPIO1_IO03;
35 static void __iomem *GPIO1_DR;
36 static void __iomem *GPIO1_GDIR;
37 
38 /* dtsled 设备结构体 */
39 struct dtsled_dev{
40     dev_t devid;              /* 设备号      */
41     struct cdev cdev;         /* cdev        */
42     struct class *class;      /* 类          */
43     struct device *device;    /* 设备        */
44     int major;                /* 主设备号    */
45     int minor;                /* 次设备号    */
46     struct device_node  *nd; /* 设备节点    */
47 };
48 
49 struct dtsled_dev dtsled;   /* led 设备 */
50 
51 /*
52  * @description   : LED 打开/关闭
53  * @param - sta   : LEDON(0) 打开 LED，LEDOFF(1) 关闭 LED
54  * @return        : 无
55  */
```

```
56 void led_switch(u8 sta)
57 {
58     u32 val = 0;
59     if(sta == LEDON) {
60         val = readl(GPIO1_DR);
61         val &= ~(1 << 3);
62         writel(val, GPIO1_DR);
63     }else if(sta == LEDOFF) {
64         val = readl(GPIO1_DR);
65         val|= (1 << 3);
66         writel(val, GPIO1_DR);
67     }
68 }
69
70 /*
71  * @description   : 打开设备
72  * @param – inode : 传递给驱动的 inode
73  * @param – filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
74  *                 一般在 open 的时候将 private_data 指向设备结构体。
75  * @return        : 0 成功;其他 失败
76  */
77 static int led_open(struct inode *inode, struct file *filp)
78 {
79     filp->private_data = &dtsled; /* 设置私有数据 */
80     return 0;
81 }
82
83 /*
84  * @description   : 从设备读取数据
85  * @param – filp  : 要打开的设备文件(文件描述符)
86  * @param - buf   : 返回给用户空间的数据缓冲区
87  * @param - cnt   : 要读取的数据长度
88  * @param – offt  : 相对于文件首地址的偏移
89  * @return        : 读取的字节数，如果为负值，表示读取失败
90  */
91 static ssize_t led_read(struct file *filp, char __user *buf, size_t
cnt, loff_t *offt)
92 {
93     return 0;
94 }
95
96 /*
97  * @description   : 向设备写数据
```

```
98   * @param - filp  : 设备文件，表示打开的文件描述符
99   * @param - buf   : 要写给设备写入的数据
100  * @param - cnt   : 要写入的数据长度
101  * @param – offt  : 相对于文件首地址的偏移
102  * @return        : 写入的字节数，如果为负值，表示写入失败
103  */
104 static ssize_t led_write(struct file *filp, const char __user *buf, 
size_t cnt, loff_t *offt)
105 {
106     int retvalue;
107     unsigned char databuf[1];
108     unsigned char ledstat;
109 
110     retvalue = copy_from_user(databuf, buf, cnt);
111     if(retvalue < 0) {
112         printk("kernel write failed!\r\n");
113         return -EFAULT;
114     }
115 
116     ledstat = databuf[0];       /* 获取状态值   */
117 
118     if(ledstat == LEDON) {
119         led_switch(LEDON);      /* 打开 LED 灯   */
120     } else if(ledstat == LEDOFF) {
121         led_switch(LEDOFF);     /* 关闭 LED 灯   */
122     }
123     return 0;
124 }
125 
126 /*
127  * @description   : 关闭/释放设备
128  * @param – filp  : 要关闭的设备文件(文件描述符)
129  * @return        : 0 成功;其他 失败
130  */
131 static int led_release(struct inode *inode, struct file *filp)
132 {
133     return 0;
134 }
135 
136 /* 设备操作函数 */
137 static struct file_operations dtsled_fops = {
138     .owner = THIS_MODULE,
139     .open = led_open,
```

```
140     .read = led_read,
141     .write = led_write,
142     .release =  led_release,
143 };
144 
145 /*
146  * @description   : 驱动入口函数
147  * @param         : 无
148  * @return        : 无
149  */
150 static int __init led_init(void)
151 {
152     u32 val = 0;
153     int ret;
154     u32 regdata[14];
155     const char *str;
156     struct property *proper;
157 
158     /* 获取设备树中的属性数据 */
159     /* 1、获取设备节点：alphaled */
160     dtsled.nd = of_find_node_by_path("/alphaled");
161     if(dtsled.nd == NULL) {
162         printk("alphaled node not find!\r\n");
163         return -EINVAL;
164     } else {
165         printk("alphaled node find!\r\n");
166     }
167 
168     /* 2、获取 compatible 属性内容 */
169     proper = of_find_property(dtsled.nd, "compatible", NULL);
170     if(proper == NULL) {
171         printk("compatible property find failed\r\n");
172     } else {
173         printk("compatible = %s\r\n", (char*)proper->value);
174     }
175 
176     /* 3、获取 status 属性内容 */
177     ret = of_property_read_string(dtsled.nd, "status", &str);
178     if(ret < 0){
179         printk("status read failed!\r\n");
180     } else {
181         printk("status = %s\r\n",str);
182     }
```

```
183
184     /* 4、获取 reg 属性内容 */
185     ret = of_property_read_u32_array(dtsled.nd, "reg", regdata, 10);
186     if(ret < 0) {
187         printk("reg property read failed!\r\n");
188     } else {
189         u8 i = 0;
190         printk("reg data:\r\n");
191         for(i = 0; i < 10; i++)
192             printk("%#X ", regdata[i]);
193         printk("\r\n");
194     }
195
196     /* 初始化 LED */
197 #if 0
198     /* 1、寄存器地址映射 */
199     IMX6U_CCM_CCGR1 = ioremap(regdata[0], regdata[1]);
200     SW_MUX_GPIO1_IO03 = ioremap(regdata[2], regdata[3]);
201     SW_PAD_GPIO1_IO03 = ioremap(regdata[4], regdata[5]);
202     GPIO1_DR = ioremap(regdata[6], regdata[7]);
203     GPIO1_GDIR = ioremap(regdata[8], regdata[9]);
204 #else
205     IMX6U_CCM_CCGR1 = of_iomap(dtsled.nd, 0);
206     SW_MUX_GPIO1_IO03 = of_iomap(dtsled.nd, 1);
207     SW_PAD_GPIO1_IO03 = of_iomap(dtsled.nd, 2);
208     GPIO1_DR = of_iomap(dtsled.nd, 3);
209     GPIO1_GDIR = of_iomap(dtsled.nd, 4);
210 #endif
211
212     /* 2、使能 GPIO1 时钟 */
213     val = readl(IMX6U_CCM_CCGR1);
214     val &= ~(3 << 26);  /* 清楚以前的设置 */
215     val |= (3 << 26);   /* 设置新值 */
216     writel(val, IMX6U_CCM_CCGR1);
217
218     /* 3、设置 GPIO1_IO03 的复用功能，将其复用为
219      *    GPIO1_IO03，最后设置 IO 属性。
220      */
221     writel(5, SW_MUX_GPIO1_IO03);
222
223     /* 寄存器 SW_PAD_GPIO1_IO03 设置 IO 属性 */
224     writel(0x10B0, SW_PAD_GPIO1_IO03);
225
```

```
226     /* 4、设置 GPIO1_IO03 为输出功能 */
227     val = readl(GPIO1_GDIR);
228     val &= ~(1 << 3);   /* 清除以前的设置 */
229     val |= (1 << 3);    /* 设置为输出 */
230     writel(val, GPIO1_GDIR);
231
232     /* 5、默认关闭 LED */
233     val = readl(GPIO1_DR);
234     val |= (1 << 3);
235     writel(val, GPIO1_DR);
236
237     /* 注册字符设备驱动 */
238     /* 1、创建设备号 */
239     if (dtsled.major) {          /*  定义了设备号 */
240         dtsled.devid = MKDEV(dtsled.major, 0);
241         register_chrdev_region(dtsled.devid, DTSLED_CNT,
                                   DTSLED_NAME);
242     } else {                      /* 没有定义设备号 */
243         alloc_chrdev_region(&dtsled.devid, 0, DTSLED_CNT,
                                 DTSLED_NAME); /* 申请设备号 */
244         dtsled.major = MAJOR(dtsled.devid); /* 获取分配号的主设备号 */
245         dtsled.minor = MINOR(dtsled.devid); /* 获取分配号的次设备号 */
246     }
247     printk("dtsled major=%d,minor=%d\r\n",dtsled.major,
            dtsled.minor);
248
249     /* 2、初始化 cdev */
250     dtsled.cdev.owner = THIS_MODULE;
251     cdev_init(&dtsled.cdev, &dtsled_fops);
252
253     /* 3、添加一个 cdev */
254     cdev_add(&dtsled.cdev, dtsled.devid, DTSLED_CNT);
255
256     /* 4、创建类 */
257     dtsled.class = class_create(THIS_MODULE, DTSLED_NAME);
258     if (IS_ERR(dtsled.class)) {
259         return PTR_ERR(dtsled.class);
260     }
261
262     /* 5、创建设备 */
263     dtsled.device = device_create(dtsled.class, NULL, dtsled.devid,
                                  NULL, DTSLED_NAME);
264     if (IS_ERR(dtsled.device)) {
```

```
265         return PTR_ERR(dtsled.device);
266     }
267 
268     return 0;
269 }
270 
271 /*
272  * @description   : 驱动出口函数
273  * @param         : 无
274  * @return        : 无
275  */
276 static void __exit led_exit(void)
277 {
278     /* 取消映射 */
279     iounmap(IMX6U_CCM_CCGR1);
280     iounmap(SW_MUX_GPIO1_IO03);
281     iounmap(SW_PAD_GPIO1_IO03);
282     iounmap(GPIO1_DR);
283     iounmap(GPIO1_GDIR);
284 
285     /* 注销字符设备驱动 */
286     cdev_del(&dtsled.cdev);/* 删除 cdev */
287     unregister_chrdev_region(dtsled.devid, DTSLED_CNT);/*注销设备号*/
288 
289     device_destroy(dtsled.class, dtsled.devid);
290     class_destroy(dtsled.class);
291 }
292 
293 module_init(led_init);
294 module_exit(led_exit);
295 MODULE_LICENSE("GPL");
296 MODULE_AUTHOR("zuozhongkai");
```

dtsled.c 文件中的内容和第四十二章的 newchrled.c 文件中的内容基本一样，只是 dtsled.c 中包含了处理设备树的代码，我们重点来看一下这部分代码。

第 46 行，在设备结构体 dtsled_dev 中添加了成员变量 nd，nd 是 device_node 结构体类型指针，表示设备节点。如果我们要读取设备树某个节点的属性值，首先要先得到这个节点，一般在设备结构体中添加 device_node 指针变量来存放这个节点。

第 160~166 行，通过 of_find_node_by_path 函数得到 alphaled 节点，后续其他的 OF 函数要使用 device_node。

第 169~174 行，通过 of_find_property 函数获取 alphaled 节点的 compatible 属性，返回值为 property 结构体类型指针变量，property 的成员变量 value 表示属性值。

第 177~182 行，通过 of_property_read_string 函数获取 alphaled 节点的 status 属性值。

第 185~194 行，通过 of_property_read_u32_array 函数获取 alphaled 节点的 reg 属性所有值，并且将获取到的值都存放到 regdata 数组中。第 192 行将获取到的 reg 属性值依次输出到终端上。

第 199~203 行，使用“古老”的 ioremap 函数完成内存映射，将获取到的 regdata 数组中的寄存器物理地址转换为虚拟地址。

第 205~209 行，使用 of_iomap 函数一次性完成读取 reg 属性以及内存映射，of_iomap 函数是设备树推荐使用的 OF 函数。

44.3.3 编写测试 APP

本章直接使用第四十二章的测试 APP，将上一章的 ledApp.c 文件复制到本章实验工程下即可。

44.4 运行测试

44.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 dtsled.o，Makefile 内容如下所示：

示例代码 44.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                 rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := dtsled.o.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 dtsled.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“dtsled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

44.4.2 运行测试

将上一小节编译出来的 dtsled.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 dtsled.ko 驱动模块：

```
depmod               //第一次加载驱动的时候需要运行此命令
modprobe dtsled.ko   //加载驱动
```

驱动加载成功以后会在终端中输出一些信息，如图 44.4.2.1 所示：

```
/lib/modules/4.1.15 # depmod
/lib/modules/4.1.15 # modprobe dtsled.ko
alphaled node find!
compatible = atkalpha-led
status = okay
reg data:
0X20C406C 0X4 0X20E0068 0X4 0X20E02F4 0X4 0X209C000 0X4 0X209C004 0X4
dtsled major=249,minor=0
/lib/modules/4.1.15 #
```

图 44.4.2.1 驱动加载成功以后输出的信息

从图 44.4.2.1 可以看出，alpahled 这个节点找到了，并且 compatible 属性值为"atkalpha-led"，status 属性值为"okay"，reg 属性的值为"0X20C406C 0X4 0X20E0068 0X4 0X20E02F4 0X4 0X209C000 0X4 0X209C004 0X4"，这些都和我们设置的设备树一致。

驱动加载成功以后就可以使用 ledApp 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/dtsled 1          //打开 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否点亮，如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp  /dev/dtsled 0         //关闭 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可：

```
rmmod dtsled.ko
```

第四十五章 pinctrl 和 gpio 子系统实验

上一章我们编写了基于设备树的 LED 驱动，但是驱动的本质还是没变，都是配置 LED 灯所使用的 GPIO 寄存器，驱动开发方式和裸机基本没啥区别。Linux 是一个庞大而完善的系统，尤其是驱动框架，像 GPIO 这种最基本的驱动不可能采用“原始”的裸机驱动开发方式，否则就相当于你买了一辆车，结果每天推着车去上班。Linux 内核提供了 pinctrl 和 gpio 子系统用于 GPIO 驱动，本章我们就来学习一下如何借助 pinctrl 和 gpio 子系统来简化 GPIO 驱动开发。

45.1 pinctrl 子系统

45.1.1 pinctrl 子系统简介

Linux 驱动讲究驱动分离与分层，pinctrl 和 gpio 子系统就是驱动分离与分层思想下的产物，驱动分离与分层其实就是按照面向对象编程的设计思想而设计的设备驱动框架，关于驱动的分离与分层我们后面会讲。本来 pinctrl 和 gpio 子系统应该放到驱动分离与分层章节后面讲解，但是不管什么外设驱动，GPIO 驱动基本都是必须的，而 pinctrl 和 gpio 子系统又是 GPIO 驱动必须使用的，所以就将 pintrcl 和 gpio 子系统这一章节提前了。

我们先来回顾一下上一章是怎么初始化 LED 灯所使用的 GPIO，步骤如下：

①、修改设备树，添加相应的节点，节点里面重点是设置 reg 属性，reg 属性包括了 GPIO 相关寄存器。

②、获取 reg 属性中 IOMUXC_SW_MUX_CTL_PAD_GPIO1_IO03 和 (IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03 这两个寄存器地址，并且初始化这两个寄存器，这两个寄存器用于设置 GPIO1_IO03 这个 PIN 的复用功能、上下拉、速度等。

③、在②里面将 GPIO1_IO03 这个 PIN 复用为了 GPIO 功能，因此需要设置 GPIO1_IO03 这个 GPIO 相关的寄存器，也就是 GPIO1_DR 和 GPIO1_GDIR 这两个寄存器。

总结一下，②中完成对 GPIO1_IO03 这个 PIN 的初始化，设置这个 PIN 的复用功能、上下拉等，比如讲 GPIO_IO03 这个 PIN 设置为 GPIO 功能。③中完成对 GPIO 的初始化，设置 GPIO 为输入/输出等。如果使用过 STM32 的话应该都记得，STM32 也是要先设置某个 PIN 的复用功能、速度、上下拉等，然后再设置 PIN 所对应的 GPIO。其实对于大多数的 32 位 SOC 而言，引脚的设置基本都是这两方面，因此 Linux 内核针对 PIN 的配置推出了 pinctrl 子系统，对于 GPIO 的配置推出了 gpio 子系统。本节我们来学习 pinctrl 子系统，下一节再学习 gpio 子系统。

大多数 SOC 的 pin 都是支持复用复用的，比如 I.MX6ULL 的 GPIO1_IO03 既可以作为普通的 GPIO 使用，也可以作为 I2C1 的 SDA 等等。此外我们还需要配置 pin 的电气特性，比如上/下拉、速度、驱动能力等等。传统的配置 pin 的方式就是直接操作相应的寄存器，但是这种配置方式比较繁琐、而且容易出问题(比如 pin 功能冲突)。pinctrl 子系统就是为了解决这个问题而引入的，pinctrl 子系统主要工作内容如下：

①、获取设备树中 pin 信息。

②、根据获取到的 pin 信息来设置 pin 的复用功能

③、根据获取到的 pin 信息来设置 pin 的电气特性，比如上/下拉、速度、驱动能力等。

对于我们使用者来讲，只需要在设备树里面设置好某个 pin 的相关属性即可，其他的初始化工作均由 pinctrl 子系统来完成，pinctrl 子系统源码目录为 drivers/pinctrl。

45.1.2 I.MX6ULL 的 pinctrl 子系统驱动

1、PIN 配置信息详解

要使用 pinctrl 子系统，我们需要在设备树里面设置 PIN 的配置信息，毕竟 pinctrl 子系统要根据你提供的信息来配置 PIN 功能，一般会在设备树里面创建一个节点来描述 PIN 的配置信息。打开 imx6ull.dtsi 文件，找到一个叫做 iomuxc 的节点，如下所示：

示例代码 45.1.2.1 iomuxc 节点内容 1

```
756 iomuxc: iomuxc@020e0000 {
757         compatible = "fsl,imx6ul-iomuxc";
758         reg = <0x020e0000 0x4000>;
```

```
759         };
```

iomuxc 节点就是 I.MX6ULL 的 IOMUXC 外设对应的节点，看起来内容很少，没看出什么跟 PIN 的配置有关的内容啊，别急！打开 imx6ull-alientek-emmc.dts，找到如下所示内容：

示例代码 45.1.2.2 iomuxc 节点内容 2

```
311 &iomuxc {
312     pinctrl-names = "default";
313     pinctrl-0 = <&pinctrl_hog_1>;
314     imx6ul-evk {
315         pinctrl_hog_1: hoggrp-1 {
316             fsl,pins = <
317                 MX6UL_PAD_UART1_RTS_B__GPIO1_IO19       0x17059
318                 MX6UL_PAD_GPIO1_IO05__USDHC1_VSELECT    0x17059
319                 MX6UL_PAD_GPIO1_IO09__GPIO1_IO09        0x17059
320                 MX6UL_PAD_GPIO1_IO00__ANATOP_OTG1_ID    0x13058
321             >;
322         };
......
371         pinctrl_flexcan1: flexcan1grp{
372             fsl,pins = <
373                 MX6UL_PAD_UART3_RTS_B__FLEXCAN1_RX      0x1b020
374                 MX6UL_PAD_UART3_CTS_B__FLEXCAN1_TX      0x1b020
375             >;
376         };
......
587         pinctrl_wdog: wdoggrp {
588             fsl,pins = <
589                 MX6UL_PAD_LCD_RESET__WDOG1_WDOG_ANY    0x30b0
590             >;
591         };
592     };
593 };
```

示例代码 45.1.2.2 就是向 iomuxc 节点追加数据，不同的外设使用的 PIN 不同、其配置也不同，因此一个萝卜一个坑，将某个外设所使用的所有 PIN 都组织在一个子节点里面。示例代码 45.1.2.2 中 pinctrl_hog_1 子节点就是和热插拔有关的 PIN 集合，比如 USB OTG 的 ID 引脚。pinctrl_flexcan1 子节点是 flexcan1 这个外设所使用的 PIN，pinctrl_wdog 子节点是 wdog 外设所使用的 PIN。如果需要在 iomuxc 中添加我们自定义外设的 PIN，那么需要新建一个子节点，然后将这个自定义外设的所有 PIN 配置信息都放到这个子节点中。

将其与示例代码 45.1.2.1 结合起来就可以得到完成的 iomuxc 节点，如下所示：

示例代码 45.1.2.3 完整的 iomuxc 节点

```
1  iomuxc: iomuxc@020e0000 {
2      compatible = "fsl,imx6ul-iomuxc";
3      reg = <0x020e0000 0x4000>;
4      pinctrl-names = "default";
```

```
5       pinctrl-0 = <&pinctrl_hog_1>;
6       imx6ul-evk {
7           pinctrl_hog_1: hoggrp-1 {
8               fsl,pins = <
9                   MX6UL_PAD_UART1_RTS_B__GPIO1_IO19      0x17059
10                  MX6UL_PAD_GPIO1_IO05__USDHC1_VSELECT   0x17059
11                  MX6UL_PAD_GPIO1_IO09__GPIO1_IO09       0x17059
12                  MX6UL_PAD_GPIO1_IO00__ANATOP_OTG1_ID   0x13058
13              >;
......
16          };
17      };
18 };
```

第 2 行，compatible 属性值为“fsl,imx6ul-iomuxc”，前面讲解设备树的时候说过，Linux 内核会根据 compatbile 属性值来查找对应的驱动文件，所以我们在 Linux 内核源码中全局搜索字符串“fsl,imx6ul-iomuxc”就会找到 I.MX6ULL 这颗 SOC 的 pinctrl 驱动文件。稍后我们会讲解这个 pinctrl 驱动文件。

第 9~12 行，pinctrl_hog_1 子节点所使用的 PIN 配置信息，我们就以第 9 行的 UART1_RTS_B 这个 PIN 为例，讲解一下如何添加 PIN 的配置信息，UART1_RTS_B 的配置信息如下：

```
MX6UL_PAD_UART1_RTS_B__GPIO1_IO19        0x17059
```

首先说明一下，UART1_RTS_B 这个 PIN 是作为 SD 卡的检测引脚，也就是通过此 PIN 就可以检测到 SD 卡是否有插入。UART1_RTS_B 的配置信息分为两部分：MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 和 0x17059

我们重点来看一下这两部分是什么含义，前面说了，对于一个 PIN 的配置主要包括两方面，一个是设置这个 PIN 的复用功能，另一个就是设置这个 PIN 的电气特性。所以我们可以大胆的猜测 UART1_RTS_B 的这两部分配置信息一个是设置 UART1_RTS_B 的复用功能，一个是用来设置 UART1_RTS_B 的电气特性。

首先来看一下 MX6UL_PAD_UART1_RTS_B__GPIO1_IO19，这是一个宏定义，定义在文件 arch/arm/boot/dts/imx6ul-pinfunc.h 中，imx6ull.dtsi 会引用 imx6ull-pinfunc.h 这个头文件，而 imx6ull-pinfunc.h 又会引用 imx6ul-pinfunc.h 这个头文件（绕啊绕！）。从这里可以看出，可以在设备树中引用 C 语言中.h 文件中的内容。MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 的宏定义内容如下：

示例代码 45.1.2.4 UART1_RTS_B 引脚定义

```
190 #define MX6UL_PAD_UART1_RTS_B__UART1_DCE_RTS 0x0090 0x031C 0x0620
                                             0x0 0x3
191 #define MX6UL_PAD_UART1_RTS_B__UART1_DTE_CTS  0x0090 0x031C 0x0000
                                             0x0 0x0
192 #define MX6UL_PAD_UART1_RTS_B__ENET1_TX_ER    0x0090 0x031C 0x0000
                                             0x1 0x0
193 #define MX6UL_PAD_UART1_RTS_B__USDHC1_CD_B    0x0090 0x031C 0x0668
                                             0x2 0x1
194 #define MX6UL_PAD_UART1_RTS_B__CSI_DATA05     0x0090 0x031C 0x04CC
                                             0x3 0x1
```

```
195 #define MX6UL_PAD_UART1_RTS_B__ENET2_1588_EVENT1_OUT 0x0090 0x031C
                                                      0x0000 0x4 0x0
196 #define MX6UL_PAD_UART1_RTS_B__GPIO1_IO19    0x0090 0x031C 0x0000
                                                 0x5 0x0
197 #define MX6UL_PAD_UART1_RTS_B__USDHC2_CD_B   0x0090 0x031C 0x0674
                                                 0x8 0x2
```

示例代码 45.1.2.4 中一共有 8 个以“MX6UL_PAD_UART1_RTS_B”开头的宏定义，大家仔细观察应该就能发现，这 8 个宏定义分别对应 UART1_RTS_B 这个 PIN 的 8 个复用 IO。查阅《I.MX6ULL 参考手册》可以知 UART1_RTS_B 的可选复用 IO 如图 45.1.2.1 所示：

MUX_MODE	MUX Mode Select Field. Select 1 of 10 iomux modes to be used for pad: UART1_RTS_B. 0000 **ALT0** — Select mux mode: ALT0 mux port: UART1_RTS_B of instance: uart1 0001 **ALT1** — Select mux mode: ALT1 mux port: ENET1_TX_ER of instance: enet1 0010 **ALT2** — Select mux mode: ALT2 mux port: USDHC1_CD_B of instance: usdhc1 0011 **ALT3** — Select mux mode: ALT3 mux port: CSI_DATA05 of instance: csi 0100 **ALT4** — Select mux mode: ALT4 mux port: ENET2_1588_EVENT1_OUT of instance: enet2 0101 **ALT5** — Select mux mode: ALT5 mux port: GPIO1_IO19 of instance: gpio1 1000 **ALT8** — Select mux mode: ALT8 mux port: USDHC2_CD_B of instance: usdhc2 1001 **ALT9** — Select mux mode: ALT9 mux port: UART5_RTS_B of instance: uart5

图 45.1.2.1 UART1_RTS_B 引脚复用

示例代码 196 行的宏定义 MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 表示将 UART1_RTS_B 这个 IO 复用为 GPIO1_IO19。此宏定义后面跟着 5 个数字，也就是这个宏定义的具体值，如下所示：

```
0x0090 0x031C 0x0000 0x5 0x0
```

这 5 个值的含义如下所示：

```
<mux_reg   conf_reg   input_reg   mux_mode   input_val>
```

综上所述可知：

0x0090：mux_reg 寄存器偏移地址，设备树中的 iomuxc 节点就是 IOMUXC 外设对应的节点，根据其 reg 属性可知 IOMUXC 外设寄存器起始地址为 0x020e0000。因此 0x020e0000+0x0090=0x020e0090，IOMUXC_SW_MUX_CTL_PAD_UART1_RTS_B 寄存器地址正好是 0x020e0090，大家可以在《IMX6ULL 参考手册》中找到 IOMUXC_SW_MUX_CTL_PAD_UART1_RTS_B 这个寄存器的位域图，如图 45.1.2.2 所示：

32.6.20 SW_MUX_CTL_PAD_UART1_RTS_B SW MUX Control Register (IOMUXC_SW_MUX_CTL_PAD_UART1_RTS_B)

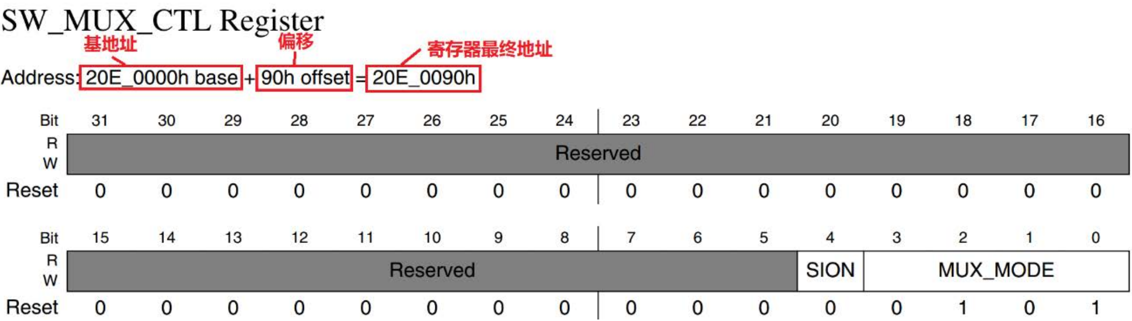


图 45.1.2.2 寄存器位域图

因此可知，0x020e0000+mux_reg 就是 PIN 的复用寄存器地址。

0x031C：conf_reg 寄存器偏移地址，和 mux_reg 一样，0x020e0000+0x031c=0x020e031c，这个就是寄存器 IOMUXC_SW_PAD_CTL_PAD_UART1_RTS_B 的地址。

0x0000：input_reg 寄存器偏移地址，有些外设有 input_reg 寄存器，有 input_reg 寄存器的外设需要配置 input_reg 寄存器。没有的话就不需要设置，UART1_RTS_B 这个 PIN 在做 GPIO1_IO19 的时候是没有 input_reg 寄存器，因此这里 intput_reg 是无效的。

0x5：mux_reg 寄存器值，在这里就相当于设置 IOMUXC_SW_MUX_CTL_PAD_UART1_RTS_B 寄存器为 0x5，也即是设置 UART1_RTS_B 这个 PIN 复用为 GPIO1_IO19。

0x0：input_reg 寄存器值，在这里无效。

这就是宏 MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 的含义，看的比较仔细的同学应该会发现并没有 conf_reg 寄存器的值，config_reg 寄存器是设置一个 PIN 的电气特性的，这么重要的寄存器怎么没有值呢？回到示例代码 45.1.2.3 中，第 9 行的内容如下所示：

```
MX6UL_PAD_UART1_RTS_B__GPIO1_IO19  0x17059
```

MX6UL_PAD_UART1_RTS_B__GPIO1_IO19 我们上面已经分析了，就剩下了一个 0x17059，反应快的同学应该已经猜出来了，0x17059 就是 conf_reg 寄存器值！此值由用户自行设置，通过此值来设置一个 IO 的上/下拉、驱动能力和速度等。在这里就相当于设置寄存器 IOMUXC_SW_PAD_CTL_PAD_UART1_RTS_B 的值为 0x17059。

2、PIN 驱动程序讲解

本小节会涉及到 Linux 驱动分层与分离、平台设备驱动等还未讲解的知识，所以本小节教程可以不用看，不会影响后续的实验。如果对 Linux 内核的 pinctrl 子系统实现原理感兴趣的话可以看本小节。

所有的东西都已经准备好了，包括寄存器地址和寄存器值，Linux 内核相应的驱动文件就会根据这些值来做相应的初始化。接下来就找一下哪个驱动文件来做这一件事情，iomuxc 节点中 compatible 属性的值为“fsl,imx6ul-iomuxc”，在 Linux 内核中全局搜索“fsl,imx6ul-iomuxc”字符串就会找到对应的驱动文件。在文件 drivers/pinctrl/freescale/pinctrl-imx6ul.c 中有如下内容：

示例代码 45.1.2.5 pinctrl-imx6ul.c 文件代码段

```
326 static struct of_device_id imx6ul_pinctrl_of_match[] = {
327     { .compatible = "fsl,imx6ul-iomuxc", .data =
                      &imx6ul_pinctrl_info, },
328     { .compatible = "fsl,imx6ull-iomuxc-snvs", .data =
                      &imx6ull_snvs_pinctrl_info, },
329     { /* sentinel */ }
330 };
331 
332 static int imx6ul_pinctrl_probe(struct platform_device *pdev)
333 {
334     const struct of_device_id *match;
335     struct imx_pinctrl_soc_info *pinctrl_info;
336 
337     match = of_match_device(imx6ul_pinctrl_of_match, &pdev->dev);
338 
339     if (!match)
```

```
340         return -ENODEV;
341 
342     pinctrl_info = (struct imx_pinctrl_soc_info *) match->data;
343 
344     return imx_pinctrl_probe(pdev, pinctrl_info);
345 }
346 
347 static struct platform_driver imx6ul_pinctrl_driver = {
348     .driver = {
349         .name = "imx6ul-pinctrl",
350         .owner = THIS_MODULE,
351         .of_match_table = of_match_ptr(imx6ul_pinctrl_of_match),
352     },
353     .probe = imx6ul_pinctrl_probe,
354     .remove = imx_pinctrl_remove,
355 };
```

第 326~330 行，of_device_id 结构体数组，第四十三章讲解设备树的时候说过了，of_device_id 里面保存着这个驱动文件的兼容性值，设备树中的 compatible 属性值会和 of_device_id 中的所有兼容性字符串比较，查看是否可以使用此驱动。imx6ul_pinctrl_of_match 结构体数组一共有两个兼容性字符串，分比为“fsl,imx6ul-iomuxc”和“fsl,imx6ull-iomuxc-snvs”，因此 iomuxc 节点与此驱动匹配，所以 pinctrl-imx6ul.c 会完成 I.MX6ULL 的 PIN 配置工作。

第 347~355 行，platform_driver 是平台设备驱动，这个是我们后面章节要讲解的内容，platform_driver 是个结构体，有个 probe 成员变量。在这里大家只需要知道，当设备和驱动匹配成功以后 platform_driver 的 probe 成员变量所代表的函数就会执行，在 353 行设置 probe 成员变量为 imx6ul_pinctrl_probe 函数，因此在本章实验中 imx6ul_pinctrl_probe 这个函数就会执行，可以认为 imx6ul_pinctrl_probe 函数就是 I.MX6ULL 这个 SOC 的 PIN 配置入口函数。以此为入口，有如图 45.1.2.3 所示的函数调用路径：

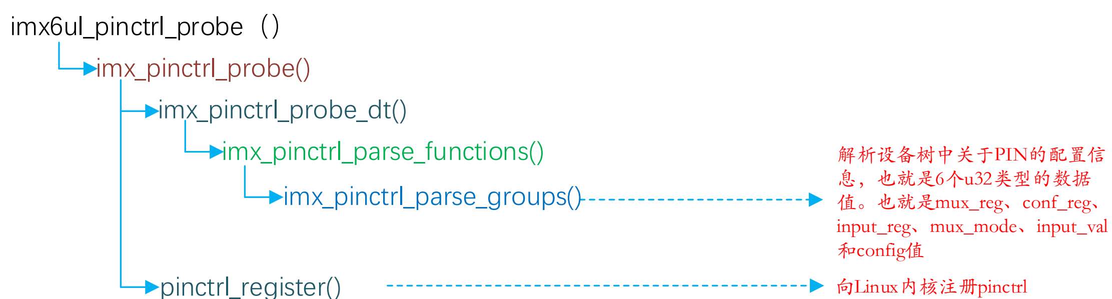


图 45.1.2.3 imx6ul_pinctrl_probe 函数执行流程

在图 45.1.2.3 中函数 imx_pinctrl_parse_groups 负责获取设备树中关于 PIN 的配置信息，也就是我们前面分析的那 6 个 u32 类型的值。处理过程如下所示：

示例代码 45.1.2.6 imx_pinctrl_parse_groups 函数代码段

```
488 /*
489  * Each pin represented in fsl,pins consists of 5 u32 PIN_FUNC_ID
490  * and 1 u32 CONFIG, so 24 types in total for each pin.
491  */
```

```
492 #define FSL_PIN_SIZE 24
493 #define SHARE_FSL_PIN_SIZE 20
494
495 static int imx_pinctrl_parse_groups(struct device_node *np,
496                 struct imx_pin_group *grp,
497                 struct imx_pinctrl_soc_info *info,
498                 u32 index)
499 {
500     int size, pin_size;
501     const __be32 *list;
502     int i;
503     u32 config;
......
537
538     for (i = 0; i < grp->npins; i++) {
539         u32 mux_reg = be32_to_cpu(*list++);
540         u32 conf_reg;
541         unsigned int pin_id;
542         struct imx_pin_reg *pin_reg;
543         struct imx_pin *pin = &grp->pins[i];
544
......
555
556         pin_id = (mux_reg != -1) ? mux_reg / 4 : conf_reg / 4;
557         pin_reg = &info->pin_regs[pin_id];
558         pin->pin = pin_id;
559         grp->pin_ids[i] = pin_id;
560         pin_reg->mux_reg = mux_reg;
561         pin_reg->conf_reg = conf_reg;
562         pin->input_reg = be32_to_cpu(*list++);
563         pin->mux_mode = be32_to_cpu(*list++);
564         pin->input_val = be32_to_cpu(*list++);
565
566         /* SION bit is in mux register */
567         config = be32_to_cpu(*list++);
568         if (config & IMX_PAD_SION)
569             pin->mux_mode |= IOMUXC_CONFIG_SION;
570         pin->config = config & ~IMX_PAD_SION;
......
574     }
575
576     return 0;
577 }
```

第 496 和 497 行，设备树中的 mux_reg 和 conf_reg 值会保存在 info 参数中，input_reg、mux_mode、input_val 和 config 值会保存在 grp 参数中。

第 560~564 行，获取 mux_reg、conf_reg、input_reg、mux_mode 和 input_val 值。

第 570 行，获取 config 值。

接下来看一下函数 pinctrl_register，此函数用于向 Linux 内核注册一个 PIN 控制器，此函数原型如下：

```
struct pinctrl_dev *pinctrl_register(struct pinctrl_desc      *pctldesc,
                                     struct device            *dev,
                                     void                     *driver_data)
```

参数 pctldesc 非常重要，因为此参数就是要注册的 PIN 控制器，PIN 控制器用于配置 SOC 的 PIN 复用功能和电气特性。参数 pctldesc 是 pinctrl_desc 结构体类型指针，pinctrl_desc 结构体如下所示：

示例代码 45.1.2.7 pinctrl_desc 结构体

```
128 struct pinctrl_desc {
129     const char *name;
130     struct pinctrl_pin_desc const *pins;
131     unsigned int npins;
132     const struct pinctrl_ops *pctlops;
133     const struct pinmux_ops *pmxops;
134     const struct pinconf_ops *confops;
135     struct module *owner;
136 #ifdef CONFIG_GENERIC_PINCONF
137     unsigned int num_custom_params;
138     const struct pinconf_generic_params *custom_params;
139     const struct pin_config_item *custom_conf_items;
140 #endif
141 };
```

第 132~124 行，这三个“_ops”结构体指针非常重要!!! 因为这三个结构体就是 PIN 控制器的“工具”，这三个结构体里面包含了很多操作函数，通过这些操作函数就可以完成对某一个 PIN 的配置。pinctrl_desc 结构体需要由用户提供，结构体里面的成员变量也是用户提供的。但是这个用户并不是我们这些使用芯片的程序员，而是半导体厂商，半导体厂商发布的 Linux 内核源码中已经把这些工作做完了。比如在 imx_pinctrl_probe 函数中可以找到如下所示代码：

示例代码 45.1.2.8 imx_pinctrl_probe 函数代码段

```
648 int imx_pinctrl_probe(struct platform_device *pdev,
649               struct imx_pinctrl_soc_info *info)
650 {
651     struct device_node *dev_np = pdev->dev.of_node;
652     struct device_node *np;
653     struct imx_pinctrl *ipctl;
654     struct resource *res;
655     struct pinctrl_desc *imx_pinctrl_desc;
......
663
```

```
664     imx_pinctrl_desc = devm_kzalloc(&pdev->dev,
                                  sizeof(*imx_pinctrl_desc),
665                               GFP_KERNEL);
666     if (!imx_pinctrl_desc)
667         return -ENOMEM;
......
705
706     imx_pinctrl_desc->name = dev_name(&pdev->dev);
707     imx_pinctrl_desc->pins = info->pins;
708     imx_pinctrl_desc->npins = info->npins;
709     imx_pinctrl_desc->pctlops = &imx_pctrl_ops;
710     imx_pinctrl_desc->pmxops = &imx_pmx_ops;
711     imx_pinctrl_desc->confops = &imx_pinconf_ops;
712     imx_pinctrl_desc->owner = THIS_MODULE;
......
723     ipctl->pctl = pinctrl_register(imx_pinctrl_desc, &pdev->dev,
                                  ipctl);
......
732 }
```

第 655 行，定义结构体指针变量 imx_pinctrl_desc。

第 664 行，向指针变量 imx_pinctrl_desc 分配内存。

第 706~712 行，初始化 imx_pinctrl_desc 结构体指针变量，重点是 pctlops、pmxops 和 confops 这三个成员变量，分别对应 imx_pctrl_ops、imx_pmx_ops 和 imx_pinconf_ops 这三个结构体。

第 723 行，调用函数 pinctrl_register 向 Linux 内核注册 imx_pinctrl_desc，注册以后 Linux 内核就有了对 I.MX6ULL 的 PIN 进行配置的工具。

imx_pctrl_ops、imx_pmx_ops 和 imx_pinconf_ops 这三个结构体定义如下：

示例代码 45.1.2.9 imx_pctrl_ops、imx_pmx_ops 和 imx_pinconf_ops 结构体

```
174 static const struct pinctrl_ops imx_pctrl_ops = {
175     .get_groups_count = imx_get_groups_count,
176     .get_group_name = imx_get_group_name,
177     .get_group_pins = imx_get_group_pins,
178     .pin_dbg_show = imx_pin_dbg_show,
179     .dt_node_to_map = imx_dt_node_to_map,
180     .dt_free_map = imx_dt_free_map,
181
182 };
......
374 static const struct pinmux_ops imx_pmx_ops = {
375     .get_functions_count = imx_pmx_get_funcs_count,
376     .get_function_name = imx_pmx_get_func_name,
377     .get_function_groups = imx_pmx_get_groups,
378     .set_mux = imx_pmx_set,
379     .gpio_request_enable = imx_pmx_gpio_request_enable,
```

```
380     .gpio_set_direction = imx_pmx_gpio_set_direction,
381 };
......
481 static const struct pinconf_ops imx_pinconf_ops = {
482     .pin_config_get = imx_pinconf_get,
483     .pin_config_set = imx_pinconf_set,
484     .pin_config_dbg_show = imx_pinconf_dbg_show,
485     .pin_config_group_dbg_show = imx_pinconf_group_dbg_show,
486 };
```

示例代码 45.1.2.9 中这三个结构体下的所有函数就是 I.MX6ULL 的 PIN 配置函数，我们就此打住，不在去分析这些函数了，否则本章就没完没了了，有兴趣的可以去看一下。

45.1.3 设备树中添加 pinctrl 节点模板

我们已经对 pinctrl 有了比较深入的了解，接下来我们学习一下如何在设备树中添加某个外设的 PIN 信息。关于 I.MX 系列 SOC 的 pinctrl 设备树绑定信息可以参考文档 Documentation/devicetree/bindings/pinctrl/fsl,imx-pinctrl.txt。这里我们虚拟一个名为“test”的设备，test 使用了 GPIO1_IO00 这个 PIN 的 GPIO 功能，pinctrl 节点添加过程如下：

1、创建对应的节点

同一个外设的 PIN 都放到一个节点里面，打开 imx6ull-alientek-emmc.dts，在 iomuxc 节点中的“imx6ul-evk”子节点下添加“pinctrl_test”节点，注意！节点前缀一定要为“pinctrl_”。添加完成以后如下所示：

示例代码 45.1.2.10 test 设备 pinctrl 节点

```
1 pinctrl_test: testgrp {
2   /* 具体的 PIN 信息 */
3 };
```

2、添加“fsl,pins”属性

设备树是通过属性来保存信息的，因此我们需要添加一个属性，属性名字一定要为“fsl,pins”，因为对于 I.MX 系列 SOC 而言，pinctrl 驱动程序是通过读取“fsl,pins”属性值来获取 PIN 的配置信息，完成以后如下所示：

示例代码 45.1.2.11 添加"fsl,pins"属性

```
1 pinctrl_test: testgrp {
2       fsl,pins = <
3       /* 设备所使用的 PIN 配置信息 */
4       >;
5 };
```

3、在“fsl,pins”属性中添加 PIN 配置信息

最后在“fsl,pins”属性中添加具体的 PIN 配置信息，完成以后如下所示：

示例代码 45.1.2.13 完整的 test 设备 pinctrl 子节点

```
1 pinctrl_test: testgrp {
2       fsl,pins = <
3       MX6UL_PAD_GPIO1_IO00__GPIO1_IO00  config /*config 是具体设置值*/
4       >;
```

```
5 };
```

至此，我们已经在 imx6ull-alientek-emmc.dts 文件中添加好了 test 设备所使用的 PIN 配置信息。

45.2 gpio 子系统

45.2.1 gpio 子系统简介

上一小节讲解了 pinctrl 子系统，pinctrl 子系统重点是设置 PIN(有的 SOC 叫做 PAD)的复用和电气属性，如果 pinctrl 子系统将一个 PIN 复用为 GPIO 的话，那么接下来就要用到 gpio 子系统了。gpio 子系统顾名思义，就是用于初始化 GPIO 并且提供提供相应的 API 函数，比如设置 GPIO 为输入输出，读取 GPIO 的值等。gpio 子系统的主要目的就是方便驱动开发者使用 gpio，驱动开发者在设备树中添加 gpio 相关信息，然后就可以在驱动程序中使用 gpio 子系统提供的 API 函数来操作 GPIO，Linux 内核向驱动开发者屏蔽掉了 GPIO 的设置过程，极大的方便了驱动开发者使用 GPIO。

45.2.2 I.MX6ULL 的 gpio 子系统驱动

1、设备树中的 gpio 信息

I.MX6ULL-ALPHA 开发板上的 UART1_RTS_B 做为 SD 卡的检测引脚，UART1_RTS_B 复用为 GPIO1_IO19，通过读取这个 GPIO 的高低电平就可以知道 SD 卡有没有插入。首先肯定是将 UART1_RTS_B 这个 PIN 复用为 GPIO1_IO19，并且设置电气属性，也就是上一小节讲的 pinctrl 节点。打开 imx6ull-alientek-emmc.dts， UART1_RTS_B 这个 PIN 的 pincrtl 设置如下：

示例代码 45.2.2.1 SD 卡 CD 引脚 PIN 配置参数

```
316 pinctrl_hog_1: hoggrp-1 {
317     fsl,pins = <
318         MX6UL_PAD_UART1_RTS_B__GPIO1_IO19   0x17059 /* SD1 CD */
......
322     >;
323 };
```

第 318 行，设置 UART1_RTS_B 这个 PIN 为 GPIO1_IO19。

pinctrl 配置好以后就是设置 gpio 了，SD 卡驱动程序通过读取 GPIO1_IO19 的值来判断 SD 卡有没有插入，但是 SD 卡驱动程序怎么知道 CD 引脚连接的 GPIO1_IO19 呢？肯定是需要设备树告诉驱动啊！在设备树中 SD 卡节点下添加一个属性来描述 SD 卡的 CD 引脚不就行了，SD 卡驱动直接读取这个属性值就知道 SD 卡的 CD 引脚使用的哪个 GPIO 了。SD 卡连接在 I.MX6ULL 的 usdhc1 接口上，在 imx6ull-alientek-emmc.dts 中找到名为“usdhc1”的节点，这个节点就是 SD 卡设备节点，如下所示：

示例代码 45.2.2.2 设备树中 SD 卡节点

```
760 &usdhc1 {
761     pinctrl-names = "default", "state_100mhz", "state_200mhz";
762     pinctrl-0 = <&pinctrl_usdhc1>;
763     pinctrl-1 = <&pinctrl_usdhc1_100mhz>;
764     pinctrl-2 = <&pinctrl_usdhc1_200mhz>;
765     /* pinctrl-3 = <&pinctrl_hog_1>; */
```

```
766     cd-gpios = <&gpio1 19 GPIO_ACTIVE_LOW>;
767     keep-power-in-suspend;
768     enable-sdio-wakeup;
769     vmmc-supply = <&reg_sd1_vmmc>;
770     status = "okay";
771 };
```

第 765 行，此行本来没有，是作者添加的，usdhc1 节点作为 SD 卡设备总节点，usdhc1 节点需要描述 SD 卡所有的信息，因为驱动要使用。本行就是描述 SD 卡的 CD 引脚 pinctrl 信息所在的子节点，因为 SD 卡驱动需要根据 pincrtl 节点信息来设置 CD 引脚的复用功能等。762~764 行的 pinctrl-0~2 都是 SD 卡其他 PIN 的 pincrtl 节点信息。但是大家会发现，其实在 usdhc1 节点中并没有"pinctrl-3 = <&pinctrl_hog_1>"这一行，也就是说并没有指定 CD 引脚的 pinctrl 信息，那么 SD 卡驱动就没法设置 CD 引脚的复用功能啊？这个不用担心，因为在"iomuxc"节点下引用了 pinctrl_hog_1 这个节点，所以 Linux 内核中的 iomuxc 驱动就会自动初始化 pinctrl_hog_1 节点下的所有 PIN。

第 766 行，属性"cd-gpios"描述了 SD 卡的 CD 引脚使用的哪个 IO。属性值一共有三个，我们来看一下这三个属性值的含义，"&gpio1"表示 CD 引脚所使用的 IO 属于 GPIO1 组，"19"表示 GPIO1 组的第 19 号 IO，通过这两个值 SD 卡驱动程序就知道 CD 引脚使用了 GPIO1_IO19 这 GPIO。"GPIO_ACTIVE_LOW"表示低电平有效，如果改为"GPIO_ACTIVE_HIGH"就表示高电平有效。

根据上面这些信息，SD 卡驱动程序就可以使用 GPIO1_IO19 来检测 SD 卡的 CD 信号了，打开 imx6ull.dtsi，在里面找到如下所示内容：

示例代码 45.2.2.2 gpio1 节点

```
504 gpio1: gpio@0209c000 {
505     compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
506     reg = <0x0209c000 0x4000>;
507     interrupts = <GIC_SPI 66 IRQ_TYPE_LEVEL_HIGH>,
508              <GIC_SPI 67 IRQ_TYPE_LEVEL_HIGH>;
509     gpio-controller;
510     #gpio-cells = <2>;
511     interrupt-controller;
512     #interrupt-cells = <2>;
513 };
```

gpio1 节点信息描述了 GPIO1 控制器的所有信息，重点就是 GPIO1 外设寄存器基地址以及兼容属性。关于 I.MX 系列 SOC 的 GPIO 控制器绑定信息请查看文档 Documentation/devicetree/bindings/gpio/ fsl-imx-gpio.txt。

第 505 行，设置 gpio1 节点的 compatible 属性有两个，分别为"fsl,imx6ul-gpio"和"fsl,imx35-gpio"，在 Linux 内核中搜索这两个字符串就可以找到 I.MX6UL 的 GPIO 驱动程序。

第 506 行，的 reg 属性设置了 GPIO1 控制器的寄存器基地址为 0X0209C000，大家可以打开《I.MX6ULL 参考手册》找到"Chapter 28:General Purpose Input/Output(GPIO)"章节第 28.5 小节，有如图 45.2.2.1 所示的寄存器地址表：

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
209_C000	GPIO data register (GPIO1_DR)	32	R/W	0000_0000h	28.5.1/1358
209_C004	GPIO direction register (GPIO1_GDIR)	32	R/W	0000_0000h	28.5.2/1359
209_C008	GPIO pad status register (GPIO1_PSR)	32	R	0000_0000h	28.5.3/1359
209_C00C	GPIO interrupt configuration register1 (GPIO1_ICR1)	32	R/W	0000_0000h	28.5.4/1360
209_C010	GPIO interrupt configuration register2 (GPIO1_ICR2)	32	R/W	0000_0000h	28.5.5/1364
209_C014	GPIO interrupt mask register (GPIO1_IMR)	32	R/W	0000_0000h	28.5.6/1367
209_C018	GPIO interrupt status register (GPIO1_ISR)	32	w1c	0000_0000h	28.5.7/1368
209_C01C	GPIO edge select register (GPIO1_EDGE_SEL)	32	R/W	0000_0000h	28.5.8/1369

图 45.2.2.1 GPIO1 寄存器表

从图 45.2.2.1 可以看出，GPIO1 控制器的基地址就是 0X0209C000。

第 509 行，“gpio-controller”表示 gpio1 节点是个 GPIO 控制器。

第 510 行，“#gpio-cells”属性和“#address-cells”类似，#gpio-cells 应该为 2，表示一共有两个 cell，第一个 cell 为 GPIO 编号，比如“&gpio1 3”就表示 GPIO1_IO03。第二个 cell 表示 GPIO 极性如果为 0 的话表示高电平有效，如果为 1 的话表示低电平有效。

2、GPIO 驱动程序简介

本小节会涉及到 Linux 驱动分层与分离、平台设备驱动等还未讲解的知识，所以本小节教程可以不用看，不会影响后续的实验。如果对 Linux 内核的 GPIO 子系统实现原理感兴趣的话可以看本小节。

gpio1 节点的 compatible 属性描述了兼容性，在 Linux 内核中搜索“fsl,imx6ul-gpio”和“fsl,imx35-gpio”这两个字符串，查找 GPIO 驱动文件。drivers/gpio/gpio-mxc.c 就是 I.MX6ULL 的 GPIO 驱动文件，在此文件中有如下所示 of_device_id 匹配表：

示例代码 45.2.2.3 mxc_gpio_dt_ids 匹配表

```
152 static const struct of_device_id mxc_gpio_dt_ids[] = {
153     { .compatible = "fsl,imx1-gpio", .data =
          &mxc_gpio_devtype[IMX1_GPIO], },
154     { .compatible = "fsl,imx21-gpio", .data =
          &mxc_gpio_devtype[IMX21_GPIO], },
155     { .compatible = "fsl,imx31-gpio", .data =
          &mxc_gpio_devtype[IMX31_GPIO], },
156     { .compatible = "fsl,imx35-gpio", .data =
          &mxc_gpio_devtype[IMX35_GPIO], },
157     { /* sentinel */ }
158 };
```

第 156 行的 compatible 值为“fsl,imx35-gpio”，和 gpio1 的 compatible 属性匹配，因此 gpio-mxc.c 就是 I.MX6ULL 的 GPIO 控制器驱动文件。gpio-mxc.c 所在的目录为 drivers/gpio，打开这个目录可以看到很多芯片的 gpio 驱动文件， “gpiolib”开始的文件是 gpio 驱动的核心文件，如图 45.2.2.2 所示：

gpiolib.c	2019-05-25 10:26	sourceinsight.c_file
gpiolib.h	2019-05-25 10:26	H 文件
gpiolib-acpi.c	2019-05-25 10:26	sourceinsight.c_file
gpiolib-legacy.c	2019-05-25 10:26	sourceinsight.c_file
gpiolib-of.c	2019-05-25 10:26	sourceinsight.c_file
gpiolib-sysfs.c	2019-05-25 10:26	sourceinsight.c_file

图 45.2.2.2 gpio 核心驱动文件

我们重点来看一下 gpio-mxc.c 这个文件，在 gpio-mxc.c 文件中有如下所示内容：

示例代码 45.2.2.4 mxc_gpio_driver 结构体

```
496 static struct platform_driver mxc_gpio_driver = {
497     .driver     = {
498         .name   = "gpio-mxc",
499         .of_match_table = mxc_gpio_dt_ids,
500     },
501     .probe      = mxc_gpio_probe,
502     .id_table   = mxc_gpio_devtype,
503 };
```

可以看出 GPIO 驱动也是个平台设备驱动，因此当设备树中的设备节点与驱动的 of_device_id 匹配以后 probe 函数就会执行，在这里就是 mxc_gpio_probe 函数，这个函数就是 I.MX6ULL 的 GPIO 驱动入口函数。我们简单来分析一下 mxc_gpio_probe 这个函数，函数内容如下：

示例代码 45.2.2.5 mxc_gpio_probe 函数

```
403 static int mxc_gpio_probe(struct platform_device *pdev)
404 {
405     struct device_node *np = pdev->dev.of_node;
406     struct mxc_gpio_port *port;
407     struct resource *iores;
408     int irq_base;
409     int err;
410 
411     mxc_gpio_get_hw(pdev);
412 
413     port = devm_kzalloc(&pdev->dev, sizeof(*port), GFP_KERNEL);
414     if (!port)
415         return -ENOMEM;
416 
417     iores = platform_get_resource(pdev, IORESOURCE_MEM, 0);
418     port->base = devm_ioremap_resource(&pdev->dev, iores);
419     if (IS_ERR(port->base))
420         return PTR_ERR(port->base);
421 
422     port->irq_high = platform_get_irq(pdev, 1);
423     port->irq = platform_get_irq(pdev, 0);
424     if (port->irq < 0)
```

```
425         return port->irq;
426 
427     /* disable the interrupt and clear the status */
428     writel(0, port->base + GPIO_IMR);
429     writel(~0, port->base + GPIO_ISR);
430 
431     if (mxc_gpio_hwtype == IMX21_GPIO) {
432         /*
433          * Setup one handler for all GPIO interrupts. Actually
434          * setting the handler is needed only once, but doing it for
435          * every port is more robust and easier.
436          */
437         irq_set_chained_handler(port->irq, mx2_gpio_irq_handler);
438     } else {
439         /* setup one handler for each entry */
440         irq_set_chained_handler(port->irq, mx3_gpio_irq_handler);
441         irq_set_handler_data(port->irq, port);
442         if (port->irq_high > 0) {
443             /* setup handler for GPIO 16 to 31 */
444             irq_set_chained_handler(port->irq_high,
445                           mx3_gpio_irq_handler);
446             irq_set_handler_data(port->irq_high, port);
447         }
448     }
449 
450     err = bgpio_init(&port->bgc, &pdev->dev, 4,
451              port->base + GPIO_PSR,
452              port->base + GPIO_DR, NULL,
453              port->base + GPIO_GDIR, NULL, 0);
454     if (err)
455         goto out_bgio;
456 
457     port->bgc.gc.to_irq = mxc_gpio_to_irq;
458     port->bgc.gc.base = (pdev->id < 0) ? of_alias_get_id(np, "gpio")
459                           * 32 : pdev->id * 32;
460 
461     err = gpiochip_add(&port->bgc.gc);
462     if (err)
463         goto out_bgpio_remove;
464 
465     irq_base = irq_alloc_descs(-1, 0, 32, numa_node_id());
466     if (irq_base < 0) {
467         err = irq_base;
```

```
468         goto out_gpiochip_remove;
469     }
470
471     port->domain = irq_domain_add_legacy(np, 32, irq_base, 0,
472                         &irq_domain_simple_ops, NULL);
473     if (!port->domain) {
474         err = -ENODEV;
475         goto out_irqdesc_free;
476     }
477
478     /* gpio-mxc can be a generic irq chip */
479     mxc_gpio_init_gc(port, irq_base);
480
481     list_add_tail(&port->node, &mxc_gpio_ports);
482
483     return 0;
......
494 }
```

第 405 行，设备树节点指针。

第 406 行，定义一个结构体指针 port，结构体类型为 mxc_gpio_port。gpio-mxc.c 的重点工作就是维护 mxc_gpio_port，mxc_gpio_port 就是对 I.MX6ULL GPIO 的抽象。mxc_gpio_port 结构体定义如下：

示例代码 45.2.2.6 mxc_gpio_port 结构体

```
61 struct mxc_gpio_port {
62     struct list_head node;
63     void __iomem *base;
64     int irq;
65     int irq_high;
66     struct irq_domain *domain;
67     struct bgpio_chip bgc;
68     u32 both_edges;
69 };
```

mxc_gpio_port 的 bgc 成员变量很重要，因为稍后的重点就是初始化 bgc。

继续回到 mxc_gpio_probe 函数函数，第 411 行调用 mxc_gpio_get_hw 函数获取 gpio 的硬件相关数据，其实就是 gpio 的寄存器组，函数 mxc_gpio_get_hw 里面有如下代码：

示例代码 45.2.2.7 mxc_gpio_get_hw 函数

```
364 static void mxc_gpio_get_hw(struct platform_device *pdev)
365 {
366     const struct of_device_id *of_id =
367             of_match_device(mxc_gpio_dt_ids, &pdev->dev);
368     enum mxc_gpio_hwtype hwtype;
......
383
```

```
384     if (hwtype == IMX35_GPIO)
385         mxc_gpio_hwdata = &imx35_gpio_hwdata;
386     else if (hwtype == IMX31_GPIO)
387         mxc_gpio_hwdata = &imx31_gpio_hwdata;
388     else
389         mxc_gpio_hwdata = &imx1_imx21_gpio_hwdata;
390 
391     mxc_gpio_hwtype = hwtype;
392 }
```

注意第 385 行，mxc_gpio_hwdata 是个全局变量，如果硬件类型是 IMX35_GPIO 的话设置 mxc_gpio_hwdat 为 imx35_gpio_hwdata。对于 I.MX6ULL 而言，硬件类型就是 IMX35_GPIO，imx35_gpio_hwdata 是个结构体变量，描述了 GPIO 寄存器组，内容如下：

示例代码 45.2.2.8 imx35_gpio_hwdata 结构体

```
101 static struct mxc_gpio_hwdata imx35_gpio_hwdata = {
102     .dr_reg       = 0x00,
103     .gdir_reg     = 0x04,
104     .psr_reg      = 0x08,
105     .icr1_reg     = 0x0c,
106     .icr2_reg     = 0x10,
107     .imr_reg      = 0x14,
108     .isr_reg      = 0x18,
109     .edge_sel_reg = 0x1c,
110     .low_level    = 0x00,
111     .high_level   = 0x01,
112     .rise_edge    = 0x02,
113     .fall_edge    = 0x03,
114 };
```

大家将 imx35_gpio_hwdata 中的各个成员变量和图 45.2.2.1 中的 GPIO 寄存器表对比就会发现，imx35_gpio_hwdata 结构体就是 GPIO 寄存器组结构。这样我们后面就可以通过 mxc_gpio_hwdata 这个全局变量来访问 GPIO 的相应寄存器了。

继续回到示例代码 45.2.2.5 的 mxc_gpio_probe 函数中，第 417 行，调用函数 platform_get_resource 获取设备树中内存资源信息，也就是 reg 属性值。前面说了 reg 属性指定了 GPIO1 控制器的寄存器基地址为 0X0209C000，在配合前面已经得到的 mxc_gpio_hwdata，这样 Linux 内核就可以访问 gpio1 的所有寄存器了。

第 418 行，调用 devm_ioremap_resource 函数进行内存映射，得到 0x0209C000 在 Linux 内核中的虚拟地址。

第 422、423 行，通过 platform_get_irq 函数获取中断号，第 422 行获取高 16 位 GPIO 的中断号，第 423 行获取底 16 位 GPIO 中断号。

第 428、429 行，操作 GPIO1 的 IMR 和 ISR 这两个寄存器，关闭 GPIO1 所有 IO 中断，并且清除状态寄存器。

第 438~448 行，设置对应 GPIO 的中断服务函数，不管是高 16 位还是低 16 位，中断服务函数都是 mx3_gpio_irq_handler。

第 450~453 行，bgpio_init 函数第一个参数为 bgc，是 bgpio_chip 结构体指针。bgpio_chip

结构体有个 gc 成员变量，gc 是个 gpio_chip 结构体类型的变量。gpio_chip 结构体是抽象出来的 GPIO 控制器，gpio_chip 结构体如下所示(有缩减)：

示例代码 45.2.2.9 gpio_chip 结构体

```
74  struct gpio_chip {
75      const char        *label;
76      struct device     *dev;
77      struct module     *owner;
78      struct list_head  list;
79  
80      int               (*request)(struct gpio_chip *chip,
81                                  unsigned offset);
82      void              (*free)(struct gpio_chip *chip,
83                                  unsigned offset);
84      int               (*get_direction)(struct gpio_chip *chip,
85                                  unsigned offset);
86      int               (*direction_input)(struct gpio_chip *chip,
87                                          unsigned offset);
88      int               (*direction_output)(struct gpio_chip *chip,
89                                      unsigned offset, int value);
90      int               (*get)(struct gpio_chip *chip,
91                              unsigned offset);
92      void              (*set)(struct gpio_chip *chip,
93                              unsigned offset, int value);
......
145 };
```

可以看出，gpio_chip 大量的成员都是函数，这些函数就是 GPIO 操作函数。bgpio_init 函数主要任务就是初始化 bgc->gc。bgpio_init 里面有三个 setup 函数：bgpio_setup_io、bgpio_setup_accessors 和 bgpio_setup_direction。这三个函数就是初始化 bgc->gc 中的各种有关 GPIO 的操作，比如输出，输入等等。第 451~453 行的 GPIO_PSR、GPIO_DR 和 GPIO_GDIR 都是 I.MX6ULL 的 GPIO 寄存器。这些寄存器地址会赋值给 bgc 参数的 reg_dat、reg_set、reg_clr 和 reg_dir 这些成员变量。至此，bgc 既有了对 GPIO 的操作函数，又有了 I.MX6ULL 有关 GPIO 的寄存器，那么只要得到 bgc 就可以对 I.MX6ULL 的 GPIO 进行操作。

继续回到 mxc_gpio_probe 函数，第 461 行调用函数 gpiochip_add 向 Linux 内核注册 gpio_chip，也就是 port->bgc.gc。注册完成以后我们就可以在驱动中使用 gpiolib 提供的各个 API 函数。

45.2.3 gpio 子系统 API 函数

对于驱动开发人员，设置好设备树以后就可以使用 gpio 子系统提供的 API 函数来操作指定的 GPIO，gpio 子系统向驱动开发人员屏蔽了具体的读写寄存器过程。这就是驱动分层与分离的好处，大家各司其职，做好自己的本职工作即可。gpio 子系统提供的常用的 API 函数有下面几个：

1、gpio_request 函数

gpio_request 函数用于申请一个 GPIO 管脚，在使用一个 GPIO 之前一定要使用 gpio_request 进行申请，函数原型如下：

```
int gpio_request(unsigned gpio,    const char *label)
```

函数参数和返回值含义如下：

gpio：要申请的 gpio 标号，使用 of_get_named_gpio 函数从设备树获取指定 GPIO 属性信息，此函数会返回这个 GPIO 的标号。

label：给 gpio 设置个名字。

返回值：0，申请成功；其他值，申请失败。

2、gpio_free 函数

如果不使用某个 GPIO 了，那么就可以调用 gpio_free 函数进行释放。函数原型如下：

```
void gpio_free(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要释放的 gpio 标号。

返回值：无。

3、gpio_direction_input 函数

此函数用于设置某个 GPIO 为输入，函数原型如下所示：

```
int gpio_direction_input(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要设置为输入的 GPIO 标号。

返回值：0，设置成功；负值，设置失败。

4、gpio_direction_output 函数

此函数用于设置某个 GPIO 为输出，并且设置默认输出值，函数原型如下：

```
int gpio_direction_output(unsigned gpio, int value)
```

函数参数和返回值含义如下：

gpio：要设置为输出的 GPIO 标号。

value：GPIO 默认输出值。

返回值：0，设置成功；负值，设置失败。

5、gpio_get_value 函数

此函数用于获取某个 GPIO 的值(0 或 1)，此函数是个宏，定义所示：

```
#define gpio_get_value    __gpio_get_value
int __gpio_get_value(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要获取的 GPIO 标号。

返回值：非负值，得到的 GPIO 值；负值，获取失败。

6、gpio_set_value 函数

此函数用于设置某个 GPIO 的值，此函数是个宏，定义如下

```
#define gpio_set_value    __gpio_set_value
void __gpio_set_value(unsigned gpio, int value)
```

函数参数和返回值含义如下：

gpio：要设置的 GPIO 标号。

value：要设置的值。

返回值：无

关于 gpio 子系统常用的 API 函数就讲这些，这些是我们用的最多的。

45.2.4 设备树中添加 gpio 节点模板

继续完成 45.1.3 中的 test 设备，在 45.1.3 中我们已经讲解了如何创建 test 设备的 pinctrl 节点。本节我们来学习一下如何创建 test 设备的 GPIO 节点。

1、创建 test 设备节点

在根节点“/”下创建 test 设备子节点，如下所示：

示例代码 45.2.4.1 test 设备节点

```
1 test {
2   /* 节点内容 */
3 };
```

2、添加 pinctrl 信息

在 45.1.3 中我们创建了 pinctrl_test 节点，此节点描述了 test 设备所使用的 GPIO_IO00 这个 PIN 的信息，我们要将这节点添加到 test 设备节点中，如下所示：

示例代码 45.2.4.2 向 test 节点添加 pinctrl 信息

```
1 test {
2   pinctrl-names = "default";
3   pinctrl-0 = <&pinctrl_test>;
4   /* 其他节点内容 */
5 };
```

第 2 行，添加 pinctrl-names 属性，此熟悉描述 pinctrl 名字为“default”。

第 3 行，添加 pinctrl-0 节点，此节点引用 45.1.3 中创建的 pinctrl_test 节点，表示 tset 设备的所使用的 PIN 信息保存在 pinctrl_test 节点中。

3、添加 GPIO 属性信息

我们最后需要在 test 节点中添加 GPIO 属性信息，表明 test 所使用的 GPIO 是哪个引脚，添加完成以后如下所示：

示例代码 45.2.4.3 向 test 节点添加 gpio 属性

```
1 test {
2   pinctrl-names = "default";
3   pinctrl-0 = <&pinctrl_test>;
4   gpio = <&gpio1 0 GPIO_ACTIVE_LOW>
```

```
5 };
```

第 4 行，test 设备所使用的 gpio。

关于 pinctrl 子系统和 gpio 子系统就讲解到这里，接下来就使用 pinctrl 和 gpio 子系统来驱动 I.MX6ULL-ALPHA 开发板上的 LED 灯。

45.2.5 与 gpio 相关的 OF 函数

在示例代码 45.2.4.3 中，我们定义了一个名为“gpio”的属性，gpio 属性描述了 test 这个设备所使用的 GPIO。在驱动程序中需要读取 gpio 属性内容，Linux 内核提供了几个与 GPIO 有关的 OF 函数，常用的几个 OF 函数如下所示：

1、of_gpio_named_count 函数

of_gpio_named_count 函数用于获取设备树某个属性里面定义了几个 GPIO 信息，要注意的是空的 GPIO 信息也会被统计到，比如：

```
gpios = <0
        &gpio1 1 2
        0
        &gpio2 3 4>;
```

上述代码的“gpios”节点一共定义了 4 个 GPIO，但是有 2 个是空的，没有实际的含义。通过 of_gpio_named_count 函数统计出来的 GPIO 数量就是 4 个，此函数原型如下：

```
int of_gpio_named_count(struct device_node *np, const char    *propname)
```

函数参数和返回值含义如下：

nd：设备节点。

propname：要统计的 GPIO 属性。

返回值：正值，统计到的 GPIO 数量；负值，失败。

2、of_gpio_count 函数

和 of_gpio_named_count 函数一样，但是不同的地方在于，此函数统计的是“gpios”这个属性的 GPIO 数量，而 of_gpio_named_count 函数可以统计任意属性的 GPIO 信息，函数原型如下所示：

```
int of_gpio_count(struct device_node *np)
```

函数参数和返回值含义如下：

nd：设备节点。

返回值：正值，统计到的 GPIO 数量；负值，失败。

3、of_get_named_gpio 函数

此函数获取 GPIO 编号，因为 Linux 内核中关于 GPIO 的 API 函数都要使用 GPIO 编号，此函数会将设备树中类似<&gpio5 7 GPIO_ACTIVE_LOW>的属性信息转换为对应的 GPIO 编号，此函数在驱动中使用很频繁！函数原型如下：

```
int of_get_named_gpio(struct device_node      *np,
                      const char              *propname,
                      int                     index)
```

函数参数和返回值含义如下：

nd：设备节点。

propname：包含要获取 GPIO 信息的属性名。

index：GPIO 索引，因为一个属性里面可能包含多个 GPIO，此参数指定要获取哪个 GPIO 的编号，如果只有一个 GPIO 信息的话此参数为 0。

返回值：正值，获取到的 GPIO 编号；负值，失败。

45.3 硬件原理图分析

本章实验硬件原理图参考 8.3 小节即可。

45.4 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 5_gpioled。

本章实验我们继续研究 LED 灯，在第四十四章实验中我们通过设备树向 dtsled.c 文件传递相应的寄存器物理地址，然后在驱动文件中配置寄存器。本章实验我们使用 pinctrl 和 gpio 子系统来完成 LED 灯驱动。

45.4.1 修改设备树文件

1、添加 pinctrl 节点

I.MX6U-ALPHA 开发板上的 LED 灯使用了 GPIO1_IO03 这个 PIN，打开 imx6ull-alientek-emmc.dts，在 iomuxc 节点的 imx6ul-evk 子节点下创建一个名为“pinctrl_led”的子节点，节点内容如下所示：

示例代码 45.4.1.1 GPIO1_IO03 pinctrl 节点

```
1 pinctrl_led: ledgrp {
2     fsl,pins = <
3         MX6UL_PAD_GPIO1_IO03__GPIO1_IO03        0x10B0 /* LED0 */
4     >;
5 };
```

第 3 行，将 GPIO1_IO03 这个 PIN 复用为 GPIO1_IO03，也就是 GPIO，GPI1O_IO03 这个 PIN 的电气属性值为 0X10B0，也就是设置 IOMUXC_SW_PAD_CTL_PAD_GPIO1_IO03 寄存器的值为 0X10B0。

2、添加 LED 设备节点

在根节点“/”下创建 LED 灯节点，节点名为“gpioled”，节点内容如下：

示例代码 45.4.1.2 创建 LED 灯节点

```
1 gpioled {
2     #address-cells = <1>;
3     #size-cells = <1>;
4     compatible = "atkalpha-gpioled";
5     pinctrl-names = "default";
6     pinctrl-0 = <&pinctrl_led>;
7     led-gpio = <&gpio1 3 GPIO_ACTIBE_LOW>;
8     status = "okay";
9 }
```

第 7 行，pinctrl-0 属性设置 LED 灯所使用的 PIN 对应的 pinctrl 节点。

第 8 行，led-gpio 属性指定了 LED 灯所使用的 GPIO，在这里就是 GPIO1 的 IO03，低电平有效。稍后编写驱动程序的时候会获取 led-gpio 属性的内容来得到 GPIO 编号，因为 gpio 子系统的 API 操作函数需要 GPIO 编号。

3、检查 PIN 是否被其他外设使用

这一点非常重要!!!

很多初次接触设备树的驱动开发人员很容易因为这个小问题栽了大跟头！因为我们所使用的设备树基本都是在半导体厂商提供的设备树文件基础上修改而来的，而半导体厂商提供的设备树是根据自己官方开发板编写的，很多 PIN 的配置和我们所使用的开发板不一样。比如 A 这个引脚在官方开发板接的是 I2C 的 SDA，而我们所使用的硬件可能将 A 这个引脚接到了其他的外设，比如 LED 灯上，接不同的外设，A 这个引脚的配置就不同。一个引脚一次只能实现一个功能，如果 A 引脚在设备树中配置为了 I2C 的 SDA 信号，那么 A 引脚就不能再配置为 GPIO，否则的话驱动程序在申请 GPIO 的时候就会失败。检查 PIN 有没有被其他外设使用包括两个方面：

①、检查 pinctrl 设置。

②、如果这个 PIN 配置为 GPIO 的话，检查这个 GPIO 有没有被别的外设使用。

在本章实验中 LED 灯使用的 PIN 为 GPIO1_IO03，因此先检查 GPIO_IO03 这个 PIN 有没有被其他的 pinctrl 节点使用，在 imx6ull-alientek-emmc.dts 中找到如下内容：

示例代码 45.4.1.3 pinctrl_tsc 节点

```
480 pinctrl_tsc: tscgrp {
481     fsl,pins = <
482         MX6UL_PAD_GPIO1_IO01__GPIO1_IO01    0xb0
483         MX6UL_PAD_GPIO1_IO02__GPIO1_IO02    0xb0
484         MX6UL_PAD_GPIO1_IO03__GPIO1_IO03    0xb0
485         MX6UL_PAD_GPIO1_IO04__GPIO1_IO04    0xb0
486     >;
487 };
```

pinctrl_tsc 节点是 TSC(电阻触摸屏接口)的 pinctrl 节点，从第 484 行可以看出，默认情况下 GPIO1_IO03 作为了 TSC 外设的 PIN。所以我们需要将第 484 行屏蔽掉！和 C 语言一样，在要屏蔽的内容前后加上“/*”和“*/”符号即可。其实在 I.MX6U-ALPHA 开发板上并没有用到 TSC 接口，所以第 482~485 行的内容可以全部屏蔽掉。

因为本章实验我们将 GPIO1_IO03 这个 PIN 配置为了 GPIO，所以还需要查找一下有没有其他的外设使用了 GPIO1_IO03，在 imx6ull-alientek-emmc.dts 中搜索“gpio1 3”，找到如下内容：

示例代码 45.4.1.4 tsc 节点

```
723 &tsc {
724     pinctrl-names = "default";
725     pinctrl-0 = <&pinctrl_tsc>;
726     xnur-gpio = <&gpio1 3 GPIO_ACTIVE_LOW>;
727     measure-delay-time = <0xffff>;
728     pre-charge-time = <0xfff>;
729     status = "okay";
730 };
```

tsc 是 TSC 的外设节点，从 726 行可以看出，tsc 外设也使用了 GPIO1_IO03，同样我们需要将这一行屏蔽掉。然后在继续搜索“gpio1 3”，看看除了本章的 LED 灯以外还有没有其他的地方也使用了 GPIO1_IO03，找到一个屏蔽一个。

设备树编写完成以后使用“make dtbs”命令重新编译设备树，然后使用新编译出来的 imx6ull-alientek-emmc.dtb 文件启动 Linux 系统。启动成功以后进入“/proc/device-tree”目录中查看“gpioled”节点是否存在，如果存在的话就说明设备树基本修改成功(具体还要驱动验证)，结果如图 45.4.1.1 所示：

```
/ # cd /proc/device-tree/
/sys/firmware/devicetree/base # ls
#address-cells                interrupt-controller@00a01000
#size-cells                   memory
aliases                       model
alphaled                      name
backlight                     pxp_v4l2
chosen                        regulators
clocks                        reserved-memory
compatible                    soc
cpus                          sound
gpioled                       spi4
/sys/firmware/devicetree/base #
```

gpioled子节点

图 45.4.1.1 gpio 子节点

45.4.2 LED 灯驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第四十四章实验驱动文件 dtsled.c 的基础上修改而来。新建名为“5_gpioled”文件夹，然后在 5_gpioled 文件夹里面创建 vscode 工程，工作区命名为“gpioled”。工程创建好以后新建 gpioled.c 文件，在 gpioled.c 里面输入如下内容：

示例代码 45.4.2.1 gpioled.c 驱动文件代码

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
17 /***************************************************************
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名    : gpioled.c
```

```
20 作者       : 左忠凯
21 版本       : V1.0
22 描述       : 采用 pinctrl 和 gpio 子系统驱动 LED 灯。
23 其他       : 无
24 论坛       : www.openedv.com
25 日志       : 初版 V1.0 2019/7/13 左忠凯创建
26 ***************************************************************/
27 #define GPIOLED_CNT      1            /* 设备号个数   */
28 #define GPIOLED_NAME     "gpioled"    /* 名字         */
29 #define LEDOFF           0            /* 关灯         */
30 #define LEDON            1            /* 开灯         */
31 
32 /* gpioled 设备结构体 */
33 struct gpioled_dev{
34     dev_t devid;                /* 设备号       */
35     struct cdev cdev;           /* cdev         */
36     struct class *class;        /* 类           */
37     struct device *device;      /* 设备         */
38     int major;                  /* 主设备号     */
39     int minor;                  /* 次设备号     */
40     struct device_node  *nd;    /* 设备节点     */
41     int led_gpio;               /* led 所使用的 GPIO 编号       */
42 };
43 
44 struct gpioled_dev gpioled; /* led 设备 */
45 
46 /*
47  * @description   : 打开设备
48  * @param - inode : 传递给驱动的 inode
49  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
50  *                  一般在 open 的时候将 private_data 指向设备结构体。
51  * @return        : 0 成功;其他 失败
52  */
53 static int led_open(struct inode *inode, struct file *filp)
54 {
55     filp->private_data = &gpioled; /* 设置私有数据 */
56     return 0;
57 }
58 
59 /*
60  * @description   : 从设备读取数据
61  * @param - filp  : 要打开的设备文件(文件描述符)
62  * @param - buf   : 返回给用户空间的数据缓冲区
```

```
63   * @param - cnt   : 要读取的数据长度
64   * @param - offt  : 相对于文件首地址的偏移
65   * @return        : 读取的字节数，如果为负值，表示读取失败
66   */
67  static ssize_t led_read(struct file *filp, char __user *buf,
                             size_t cnt, loff_t *offt)
68  {
69      return 0;
70  }
71
72  /*
73   * @description   : 向设备写数据
74   * @param - filp  : 设备文件，表示打开的文件描述符
75   * @param - buf   : 要写给设备写入的数据
76   * @param - cnt   : 要写入的数据长度
77   * @param - offt  : 相对于文件首地址的偏移
78   * @return        : 写入的字节数，如果为负值，表示写入失败
79   */
80  static ssize_t led_write(struct file *filp, const char __user *buf,
                             size_t cnt, loff_t *offt)
81  {
82      int retvalue;
83      unsigned char databuf[1];
84      unsigned char ledstat;
85      struct gpioled_dev *dev = filp->private_data;
86
87      retvalue = copy_from_user(databuf, buf, cnt);
88      if(retvalue < 0) {
89          printk("kernel write failed!\r\n");
90          return -EFAULT;
91      }
92
93      ledstat = databuf[0];               /* 获取状态值   */
94
95      if(ledstat == LEDON) {
96          gpio_set_value(dev->led_gpio, 0);   /* 打开 LED 灯  */
97      } else if(ledstat == LEDOFF) {
98          gpio_set_value(dev->led_gpio, 1);   /* 关闭 LED 灯  */
99      }
100     return 0;
101 }
102
103 /*
```

```
104  * @description   : 关闭/释放设备
105  * @param – filp : 要关闭的设备文件(文件描述符)
106  * @return        : 0 成功;其他 失败
107  */
108 static int led_release(struct inode *inode, struct file *filp)
109 {
110     return 0;
111 }
112
113 /* 设备操作函数 */
114 static struct file_operations gpioled_fops = {
115     .owner = THIS_MODULE,
116     .open = led_open,
117     .read = led_read,
118     .write = led_write,
119     .release =  led_release,
120 };
121
122 /*
123  * @description   : 驱动入口函数
124  * @param         : 无
125  * @return        : 无
126  */
127 static int __init led_init(void)
128 {
129     int ret = 0;
130
131     /* 设置 LED 所使用的 GPIO */
132     /* 1、获取设备节点：gpioled */
133     gpioled.nd = of_find_node_by_path("/gpioled");
134     if(gpioled.nd == NULL) {
135         printk("gpioled node not find!\r\n");
136         return -EINVAL;
137     } else {
138         printk("gpioled node find!\r\n");
139     }
140
141     /* 2、 获取设备树中的 gpio 属性，得到 LED 所使用的 LED 编号 */
142     gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
143     if(gpioled.led_gpio < 0) {
144         printk("can't get led-gpio");
145         return -EINVAL;
146     }
```

```
147     printk("led-gpio num = %d\r\n", gpioled.led_gpio);
148 
149     /* 3、设置 GPIO1_IO03 为输出，并且输出高电平，默认关闭 LED 灯 */
150     ret = gpio_direction_output(gpioled.led_gpio, 1);
151     if(ret < 0) {
152         printk("can't set gpio!\r\n");
153     }
154 
155     /* 注册字符设备驱动 */
156     /* 1、创建设备号 */
157     if (gpioled.major) {                      /*  定义了设备号    */
158         gpioled.devid = MKDEV(gpioled.major, 0);
159         register_chrdev_region(gpioled.devid, GPIOLED_CNT,
                                 GPIOLED_NAME);
160     } else {                                  /* 没有定义设备号   */
161         alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,
                           GPIOLED_NAME); /* 申请设备号       */
162         gpioled.major = MAJOR(gpioled.devid);  /* 获取分配号的主设备号 */
163         gpioled.minor = MINOR(gpioled.devid); /* 获取分配号的次设备号 */
164     }
165     printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
            gpioled.minor);
166 
167     /* 2、初始化 cdev */
168     gpioled.cdev.owner = THIS_MODULE;
169     cdev_init(&gpioled.cdev, &gpioled_fops);
170 
171     /* 3、添加一个 cdev */
172     cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
173 
174     /* 4、创建类 */
175     gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
176     if (IS_ERR(gpioled.class)) {
177         return PTR_ERR(gpioled.class);
178     }
179 
180     /* 5、创建设备 */
181     gpioled.device = device_create(gpioled.class, NULL,
                                    gpioled.devid, NULL, GPIOLED_NAME);
182     if (IS_ERR(gpioled.device)) {
183         return PTR_ERR(gpioled.device);
184     }
185     return 0;
```

```
186 }
187
188 /*
189  * @description   : 驱动出口函数
190  * @param         : 无
191  * @return        : 无
192  */
193 static void __exit led_exit(void)
194 {
195     /* 注销字符设备驱动 */
196     cdev_del(&gpioled.cdev);        /*  删除 cdev */
197     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT); /* 注销 */
198
199     device_destroy(gpioled.class, gpioled.devid);
200     class_destroy(gpioled.class);
201 }
202
203 module_init(led_init);
204 module_exit(led_exit);
205 MODULE_LICENSE("GPL");
206 MODULE_AUTHOR("zuozhongkai");
```

第 41 行，在设备结构体 gpioled_dev 中加入 led_gpio 这个成员变量，此成员变量保存 LED 等所使用的 GPIO 编号。

第 55 行，将设备结构体变量 gpioled 设置为 filp 的私有数据 private_data。

第 85 行，通过读取 filp 的 private_data 成员变量来得到设备结构体变量，也就是 gpioled。这种将设备结构体设置为 filp 私有数据的方法在 Linux 内核驱动里面非常常见。

第 96、97 行，直接调用 gpio_set_value 函数来向 GPIO 写入数据，实现开/关 LED 的效果。不需要我们直接操作相应的寄存器。

第 133 行，获取节点“/gpioled”。

第 142 行，通过函数 of_get_named_gpio 函数获取 LED 所使用的 LED 编号。相当于将 gpioled 节点中的“led-gpio”属性值转换为对应的 LED 编号。

第 150 行，调用函数 gpio_direction_output 设置 GPIO1_IO03 这个 GPIO 为输出，并且默认高电平，这样默认就会关闭 LED 灯。

可以看出 gpioled.c 文件中的内容和第四十四章的 dtsled.c 差不多，只是取消掉了配置寄存器的过程，改为使用 Linux 内核提供的 API 函数。在 GPIO 操作上更加的规范化，符合 Linux 代码框架，而且也简化了 GPIO 驱动开发的难度，以后我们所有例程用到 GPIO 的地方都采用此方法。

44.4.3 编写测试 APP

本章直接使用第四十二章的测试 APP，将上一章的 ledApp.c 文件复制到本章实验工程下即可。

45.5 运行测试

45.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 gpioled.o，Makefile 内容如下所示：

示例代码 45.5.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := gpioled.o.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 gpioled.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“gpioled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

45.5.2 运行测试

将上一小节编译出来的 gpioled.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 gpioled.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe gpioled.ko     //加载驱动
```

驱动加载成功以后会在终端中输出一些信息，如图 45.5.2.1 所示：

```
/lib/modules/4.1.15 # depmod
/lib/modules/4.1.15 # modprobe gpioled.ko
gpioled node find!
led-gpio num = 3
gpioled major=249,minor=0
/lib/modules/4.1.15 #
```

图 45.5.2.1 驱动加载成功以后输出的信息

从图 45.5.2.1 可以看出，gpioled 这个节点找到了，并且 GPIO1_IO03 这个 GPIO 的编号为 3。驱动加载成功以后就可以使用 ledApp 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/gpioled 1        //打开 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否点亮，如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp  /dev/gpioled 0    //关闭 LED 灯
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可：

```
rmmod gpioled.ko
```

第四十六章 Linux 蜂鸣器实验

上一章实验中我们借助 pinctrl 和 gpio 子系统编写了 LED 灯驱动，I.MX6U-ALPHA 开发板上还有一个蜂鸣器，从软件的角度考虑，蜂鸣器驱动和 LED 灯驱动其实是一摸一样的，都是控制 IO 输出高低电平。本章我们就来学习编写蜂鸣器的 Linux 驱动，也算是对上一章讲解的 pinctrl 和 gpio 子系统的巩固。

46.1 蜂鸣器驱动原理

蜂鸣器驱动原理已经在第十四章有了详细的讲解，I.MX6U-ALPHA 开发板上的蜂鸣器通过 SNVS_TAMPER1 引脚来控制，本节我们来看一下如果在 Linux 下编写蜂鸣器驱动需要做哪些工作：

①、在设备树中添加 SNVS_TAMPER1 引脚的 pinctrl 信息。

②、在设备树中创建蜂鸣器节点，在蜂鸣器节点中加入 GPIO 信息。

③、编写驱动程序和测试 APP，和第四十五章的 LED 驱动程序和测试 APP 基本一样。

接下来我们就根据上面这三步来编写蜂鸣器 Linux 驱动。

46.2 硬件原理图分析

本章实验硬件原理图参考 14.3 小节即可。

46.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 6_beep。

本章实验在四十二章实验的基础上完成，重点是将驱动改为基于设备树的.

46.3.1 修改设备树文件

1、添加 pinctrl 节点

I.MX6U-ALPHA 开发板上的 BEEP 使用了 SNVS_TAMPER1 这个 PIN，打开 imx6ull-alientek-emmc.dts，在 iomuxc 节点的 imx6ul-evk 子节点下创建一个名为“pinctrl_beep”的子节点，节点内容如下所示：

示例代码 46.3.1.1 SNVS_TAMPER1 pinctrl 节点

```
1 pinctrl_beep: beepgrp {
2     fsl,pins = <
3         MX6ULL_PAD_SNVS_TAMPER1__GPIO5_IO01     0x10B0 /* beep */
4     >;
5 };
```

第 3 行，将 SNVS_TAMPER1 这个 PIN 复用为 GPIO5_IO01，宏 MX6ULL_PAD_SNVS_TAMPER1__GPIO5_IO01 定义在 arch/arm/boot/dts/imx6ull-pinfunc-snvs.h 文件中。

2、添加 BEEP 设备节点

在根节点“/”下创建 BEEP 节点，节点名为“beep”，节点内容如下：

示例代码 46.3.1.2 创建 BEEP 蜂鸣器节点

```
1 beep {
2     #address-cells = <1>;
3     #size-cells = <1>;
4     compatible = "atkalpha-beep";
5     pinctrl-names = "default";
6     pinctrl-0 = <&pinctrl_beep>;
7     beep-gpio = <&gpio5 1 GPIO_ACTIVE_HIGH>;
8     status = "okay";
```

```
9 };
```

第 6 行，pinctrl-0 属性设置蜂鸣器所使用的 PIN 对应的 pinctrl 节点。

第 7 行，beep-gpio 属性指定了蜂鸣器所使用的 GPIO。

3、检查 PIN 是否被其他外设使用

在本章实验中蜂鸣器使用的 PIN 为 SNVS_TAMPER1，因此先检查 PIN 为 SNVS_TAMPER1 这个 PIN 有没有被其他的 pinctrl 节点使用，如果有使用的话就要屏蔽掉，然后再检查 GPIO5_IO01 这个 GPIO 有没有被其他外设使用，如果有的话也要屏蔽掉。

设备树编写完成以后使用“make dtbs”命令重新编译设备树，然后使用新编译出来的 imx6ull-alientek-emmc.dtb 文件启动 Linux 系统。启动成功以后进入“/proc/device-tree”目录中查看“beep”节点是否存在，如果存在的话就说明设备树基本修改成功(具体还要驱动验证)，结果如图 46.3.1.1 所示：

```
/ # ls /proc/device-tree/
#address-cells            interrupt-controller@00a01000
#size-cells               memory
aliases                   model
alphaled                  name
backlight                 pxp_v4l2
beep                      regulators
chosen                    reserved-memory
clocks                    soc
compatible                sound
cpus                      spi4
gpioled
/ #
```

beep子节点

图 46.3.1.1 beep 子节点

46.3.2 蜂鸣器驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第四十五章实验驱动文件 gpioled.c 的基础上修改而来。新建名为“6_beep”的文件夹，然后在 6_beep 文件夹里面创建 vscode 工程，工作区命名为“beep”。工程创建好以后新建 beep.c 文件，在 beep.c 里面输入如下内容：

示例代码 46.3.2.1 beep.c 文件代码段

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
```

```
17 /***************************************************************
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名       : beep.c
20 作者         : 左忠凯
21 版本         : V1.0
22 描述         : 蜂鸣器驱动程序。
23 其他         : 无
24 论坛         : www.openedv.com
25 日志         : 初版 V1.0 2019/7/15 左忠凯创建
26 ***************************************************************/
27 #define BEEP_CNT          1           /* 设备号个数    */
28 #define BEEP_NAME         "beep"      /* 名字          */
29 #define BEEPOFF           0           /* 关蜂鸣器      */
30 #define BEEPON            1           /* 开蜂鸣器      */
31
32
33 /* beep 设备结构体 */
34 struct beep_dev{
35     dev_t devid;               /* 设备号                    */
36     struct cdev cdev;          /* cdev                      */
37     struct class *class;       /* 类                        */
38     struct device *device;     /* 设备                      */
39     int major;                 /* 主设备号                  */
40     int minor;                 /* 次设备号                  */
41     struct device_node  *nd; /* 设备节点                    */
42     int beep_gpio;             /* beep 所使用的 GPIO 编号    */
43 };
44
45 struct beep_dev beep;          /* beep 设备 */
46
47 /*
48  * @description    : 打开设备
49  * @param – inode : 传递给驱动的 inode
50  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
51  *                   一般在 open 的时候将 private_data 指向设备结构体。
52  * @return         : 0 成功;其他 失败
53  */
54 static int beep_open(struct inode *inode, struct file *filp)
55 {
56     filp->private_data = &beep; /* 设置私有数据 */
57     return 0;
58 }
59
```

```
60 /*
61  * @description   : 向设备写数据
62  * @param - filp  : 设备文件，表示打开的文件描述符
63  * @param - buf   : 要写给设备写入的数据
64  * @param - cnt   : 要写入的数据长度
65  * @param - offt  : 相对于文件首地址的偏移
66  * @return        : 写入的字节数，如果为负值，表示写入失败
67  */
68 static ssize_t beep_write(struct file *filp, const char __user *buf,
                            size_t cnt, loff_t *offt)
69 {
70     int retvalue;
71     unsigned char databuf[1];
72     unsigned char beepstat;
73     struct beep_dev *dev = filp->private_data;
74 
75     retvalue = copy_from_user(databuf, buf, cnt);
76     if(retvalue < 0) {
77         printk("kernel write failed!\r\n");
78         return -EFAULT;
79     }
80 
81     beepstat = databuf[0];                  /* 获取状态值 */
82 
83     if(beepstat == BEEPON) {
84         gpio_set_value(dev->beep_gpio, 0);  /* 打开蜂鸣器 */
85     } else if(beepstat == BEEPOFF) {
86         gpio_set_value(dev->beep_gpio, 1);  /* 关闭蜂鸣器 */
87     }
88     return 0;
89 }
90 
91 /*
92  * @description   : 关闭/释放设备
93  * @param - filp  : 要关闭的设备文件(文件描述符)
94  * @return        : 0 成功;其他 失败
95  */
96 static int beep_release(struct inode *inode, struct file *filp)
97 {
98     return 0;
99 }
100 
101 /* 设备操作函数 */
```

```
102 static struct file_operations beep_fops = {
103     .owner = THIS_MODULE,
104     .open = beep_open,
105     .write = beep_write,
106     .release =  beep_release,
107 };
108 
109 /*
110  * @description   : 驱动入口函数
111  * @param         : 无
112  * @return        : 无
113  */
114 static int __init beep_init(void)
115 {
116     int ret = 0;
117 
118     /* 设置 BEEP 所使用的 GPIO */
119     /* 1、获取设备节点：beep */
120     beep.nd = of_find_node_by_path("/beep");
121     if(beep.nd == NULL) {
122         printk("beep node not find!\r\n");
123         return -EINVAL;
124     } else {
125         printk("beep node find!\r\n");
126     }
127 
128     /* 2、 获取设备树中的 gpio 属性，得到 BEEP 所使用的 GPIO 编号 */
129     beep.beep_gpio = of_get_named_gpio(beep.nd, "beep-gpio", 0);
130     if(beep.beep_gpio < 0) {
131         printk("can't get beep-gpio");
132         return -EINVAL;
133     }
134     printk("led-gpio num = %d\r\n", beep.beep_gpio);
135 
136     /* 3、设置 GPIO5_IO01 为输出，并且输出高电平，默认关闭 BEEP */
137     ret = gpio_direction_output(beep.beep_gpio, 1);
138     if(ret < 0) {
139         printk("can't set gpio!\r\n");
140     }
141 
142     /* 注册字符设备驱动 */
143     /* 1、创建设备号 */
144     if (beep.major) {                       /*  定义了设备号 */
```

```
145         beep.devid = MKDEV(beep.major, 0);
146         register_chrdev_region(beep.devid, BEEP_CNT, BEEP_NAME);
147     } else {                            /* 没有定义设备号 */
148         alloc_chrdev_region(&beep.devid, 0, BEEP_CNT, BEEP_NAME);
149         beep.major = MAJOR(beep.devid); /* 获取分配号的主设备号 */
150         beep.minor = MINOR(beep.devid); /* 获取分配号的次设备号 */
151     }
152     printk("beep major=%d,minor=%d\r\n",beep.major, beep.minor);
153 
154     /* 2、初始化 cdev */
155     beep.cdev.owner = THIS_MODULE;
156     cdev_init(&beep.cdev, &beep_fops);
157 
158     /* 3、添加一个 cdev */
159     cdev_add(&beep.cdev, beep.devid, BEEP_CNT);
160 
161     /* 4、创建类 */
162     beep.class = class_create(THIS_MODULE, BEEP_NAME);
163     if (IS_ERR(beep.class)) {
164         return PTR_ERR(beep.class);
165     }
166 
167     /* 5、创建设备 */
168     beep.device = device_create(beep.class, NULL, beep.devid, NULL,
                                 BEEP_NAME);
169     if (IS_ERR(beep.device)) {
170         return PTR_ERR(beep.device);
171     }
172 
173     return 0;
174 }
175 
176 /*
177  * @description    : 驱动出口函数
178  * @param          : 无
179  * @return         : 无
180  */
181 static void __exit beep_exit(void)
182 {
183     /* 注销字符设备驱动 */
184     cdev_del(&beep.cdev);                           /*  删除 cdev */
185     unregister_chrdev_region(beep.devid, BEEP_CNT);  /* 注销设备号 */
186 
```

```
187     device_destroy(beep.class, beep.devid);
188     class_destroy(beep.class);
189 }
190
191 module_init(beep_init);
192 module_exit(beep_exit);
193 MODULE_LICENSE("GPL");
194 MODULE_AUTHOR("zuozhongkai");
```

beep.c 中的内容和上一章的 gpioled.c 中的内容基本一样，只是换为了初始化 SNVS_TAMPER1 这个 PIN，这里就不详细的讲解了。

46.3.3 编写测试 APP

测试 APP 在上一章实验的 ledApp.c 文件的基础上完成，新建名为 beepApp.c 的文件，然后输入如下所示内容：

示例代码 46.3.3.1 beepApp.c 文件

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名         : beepApp.c
11 作者           : 左忠凯
12 版本           : V1.0
13 描述           : beep 测试 APP。
14 其他           : 无
15 使用方法       : ./beepApp /dev/beep  0 关闭蜂鸣器
16                  ./beepApp /dev/beep  1 打开蜂鸣器
17 论坛           : www.openedv.com
18 日志           : 初版 V1.0 2019/7/15 左忠凯创建
19 ***************************************************************/
20
21 #define BEEPOFF   0
22 #define BEEPON    1
23
24 /*
25  * @description    : main 主程序
26  * @param - argc   : argv 数组元素个数
27  * @param - argv   : 具体参数
28  * @return         : 0 成功;其他 失败
```

```
29  */
30 int main(int argc, char *argv[])
31 {
32     int fd, retvalue;
33     char *filename;
34     unsigned char databuf[1];
35 
36     if(argc != 3){
37         printf("Error Usage!\r\n");
38         return -1;
39     }
40 
41     filename = argv[1];
42 
43     /* 打开 beep 驱动 */
44     fd = open(filename, O_RDWR);
45     if(fd < 0){
46         printf("file %s open failed!\r\n", argv[1]);
47         return -1;
48     }
49 
50     databuf[0] = atoi(argv[2]); /* 要执行的操作：打开或关闭 */
51 
52     /* 向/dev/beep 文件写入数据 */
53     retvalue = write(fd, databuf, sizeof(databuf));
54     if(retvalue < 0){
55         printf("BEEP Control Failed!\r\n");
56         close(fd);
57         return -1;
58     }
59 
60     retvalue = close(fd); /* 关闭文件 */
61     if(retvalue < 0){
62         printf("file %s close failed!\r\n", argv[1]);
63         return -1;
64     }
65     return 0;
66 }
```

beepApp.c 的文件内容和 ledApp.c 文件内容基本一样，要是对文件进行打开、写、关闭等操作。

46.4 运行测试

46.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 beep.o，Makefile 内容如下所示：

示例代码 46.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := beep.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 beep.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“beep.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 beepApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc beepApp.c -o beepApp
```

编译成功以后就会生成 beepApp 这个应用程序。

46.4.2 运行测试

将上一小节编译出来的 beep.ko 和 beepApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 beep.ko 驱动模块：

```
depmod              //第一次加载驱动的时候需要运行此命令
modprobe beep.ko    //加载驱动
```

驱动加载成功以后会在终端中输出一些信息，如图 46.4.2.1 所示：

```
/lib/modules/4.1.15 # modprobe beep.ko
beep node find!
led-gpio num = 129
beep major=249,minor=0
```

图 46.4.2.1 驱动加载成功以后输出的信息

从图 46.4.2.1 可以看出，beep 这个节点找到了，并且 GPIO5_IO01 这个 GPIO 的编号为 129。使用 beepApp 软件来测试驱动是否工作正常，输入如下命令打开蜂鸣器：

```
./beepApp  /dev/beep 1    //打开蜂鸣器
```

输入上述命令，查看 I.MX6U-ALPHA 开发板上的蜂鸣器是否有鸣叫，如果鸣叫的话说明驱动工作正常。在输入如下命令关闭蜂鸣器：

```
./beepApp  /dev/beep 0    //关闭蜂鸣器
```

输入上述命令以后观察 I.MX6U-ALPHA 开发板上的蜂鸣器是否停止鸣叫。如果要卸载驱动的话输入如下命令即可：

```
rmmod beep.ko
```

第四十七章 Linux 并发与竞争

Linux 是一个多任务操作系统，肯定会存在多个任务共同操作同一段内存或者设备的情况，多个任务甚至中断都能访问的资源叫做共享资源，就和共享单车一样。在驱动开发中要注意对共享资源的保护，也就是要处理对共享资源的并发访问。比如共享单车，大家按照谁扫谁骑走的原则来共用这个单车，如果没有这个并发访问共享单车的原则存在，只怕到时候为了一辆单车要打起来了。在 Linux 驱动编写过程中对于并发控制的管理非常重要，本章我们就来学习一下如何在 Linux 驱动中处理并发。

47.1 并发与竞争

1、并发与竞争简介

并发就是多个“用户”同时访问同一个共享资源，比如你们公司有一台打印机，你们公司的所有人都可以使用。现在小李和小王要同时使用这一台打印机，都要打印一份文件。小李要打印的文件内容如下：

示例代码 47.1.1 小李要打印的内容

```
我叫小李
电话：123456
工号：16
```

小王要打印的内容如下：

示例代码 47.1.2 小王要打印的内容

```
我叫小王
电话：678910
工号：20
```

这两份文档肯定是各自打印出来的，不能相互影响。当两个人同时打印的话如果打印机不做处理的话可能会出现小李的文档打印了一行，然后开始打印小王的文档，这样打印出来的文档就错乱了，可能会出现如下的错误文档内容：

示例代码 47.1.3 小王打印出来的错误文档

```
我叫小王
电话：123456
工号：20
```

可以看出，小王打印出来的文档中电话号码错误了，变成小李的了，这是绝对不允许的。如果有多人同时向打印机发送了多份文档，打印机必须保证一次只能打印一份文档，只有打印完成以后才能打印其他的文档。

Linux 系统是个多任务操作系统，会存在多个任务同时访问同一片内存区域，这些任务可能会相互覆盖这段内存中的数据，造成内存数据混乱。针对这个问题必须要做处理，严重的话可能会导致系统崩溃。现在的 Linux 系统并发产生的原因很复杂，总结一下有下面几个主要原因：

①、多线程并发访问，Linux 是多任务(线程)的系统，所以多线程访问是最基本的原因。

②、抢占式并发访问，从 2.6 版本内核开始，Linux 内核支持抢占，也就是说调度程序可以在任意时刻抢占正在运行的线程，从而运行其他的线程。

③、中断程序并发访问，这个无需多说，学过 STM32 的同学应该知道，硬件中断的权利可是很大的。

④、SMP(多核)核间并发访问，现在 ARM 架构的多核 SOC 很常见，多核 CPU 存在核间并发访问。

并发访问带来的问题就是竞争，学过 FreeRTOS 和 UCOS 的同学应该知道临界区这个概念，所谓的临界区就是共享数据段，对于临界区必须保证一次只有一个线程访问，也就是要保证临界区是原子访问的，注意这里的“原子”不是正点原子的“原子”。我们都知道，原子化学反应不可再分的基本微粒，这里的原子访问就表示这一个访问是一个步骤，不能再进行拆分。如果多个线程同时操作临界区就表示存在竞争，我们在编写驱动的时候一定要注意避免并发和防止竞争访问。很多 Linux 驱动初学者往往不注意这一点，在驱动程序中埋下了隐患，这类问题往

往又很不容易查找，导致驱动调试难度加大、费时费力。所以我们一般在编写驱动的时候就要考虑到并发与竞争，而不是驱动都编写完了然后再处理并发与竞争。

2、保护内容是什么

前面一直说要防止并发访问共享资源，换句话说就是要保护共享资源，防止进行并发访问。那么问题来了，什么是共享资源？现实生活中的公共电话、共享单车这些是共享资源，我们都很容易理解，那么在程序中什么是共享资源？也就是保护的内容是什么？我们保护的不是代码，而是数据！数据！数据！某个线程的局部变量不需要保护，我们要保护的是多个线程都会访问的共享数据。一个整形的全局变量 a 是数据，一份要打印的文档也是数据，虽然我们知道了要对共享数据进行保护，那么怎么判断哪些共享数据要保护呢？找到要保护的数据才是重点，而这个也是难点，因为驱动程序各不相同，那么数据也千变万化，一般像全局变量，设备机构体这些肯定是要保护的，至于其他的数据就要根据实际的驱动程序而定了。

当我们发现驱动程序中存在并发和竞争的时候一定要处理掉，接下来我们依次来学习一下 Linux 内核提供的几种并发和竞争的处理方法。

47.2 原子操作

47.2.1 原子操作简介

首先看一下原子操作，原子操作就是指不能在进一步分割的指令，一般原子操作用于变量或者位操作。假如现在要对无符号整形变量 a 赋值，值为 3，对于 C 语言来讲很简单，直接就是：

```
a = 3
```

但是 C 语言要先编译为成汇编指令，ARM 架构不支持直接对寄存器进行读写操作，比如要借助寄存器 R0、R1 等来完成赋值操作。假设变量 a 的地址为 0X3000000，“a=3”这一行 C 语言可能会被编译为如下所示的汇编代码：

示例代码 47.2.1.1 汇编示例代码

```
1 ldr r0, =0X30000000     /* 变量 a 地址        */
2 ldr r1, = 5             /* 要写入的值         */
3 str r1, [r0]            /* 将 5 写入到 a 变量中 */
```

示例代码 47.2.1.1 只是一个简单的距离说明，实际的结果要比示例代码复杂的多。从上述代码可以看出，C 语言里面简简单单的一句“a=3”，编译成汇编文件以后变成了 3 句，那么程序在执行的时候肯定是按照示例代码 47.2.1.1 中的汇编语句一条一条的执行。假设现在线程 A 要向 a 变量写入 10 这个值，而线程 B 也要向 a 变量写入 20 这个值，我们理想中的执行顺序如图 47.2.1.1 所示：

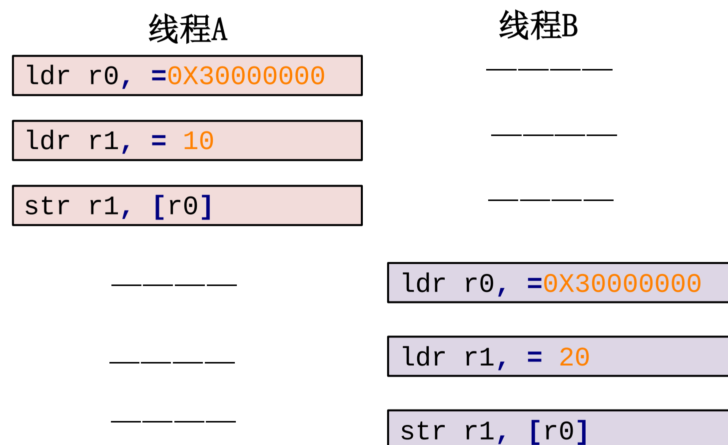


图 47.2.1.1 理想的执行流程

按照图 47.2.1.1 所示的流程，确实可以实现线程 A 将 a 变量设置为 10，线程 B 将 a 变量设置为 20。但是实际上的执行流程可能如图 47.2.1.2 所示：

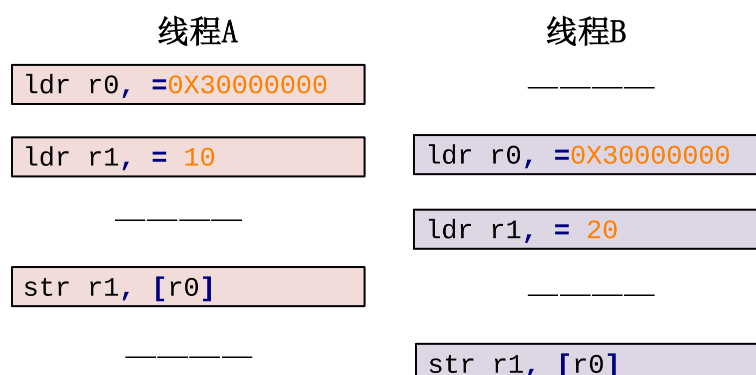


图 47.2.1.2 可能的执行流程

按照图 47.2.1.2 所示的流程，线程 A 最终将变量 a 设置为了 20，而并不是要求的 10！线程 B 没有问题。这就是一个最简单的设置变量值的并发与竞争的例子，要解决这个问题就要保证示例代码 47.2.1.1 中的三行汇编指令作为一个整体运行，也就是作为一个原子存在。Linux 内核提供了一组原子操作 API 函数来完成此功能，Linux 内核提供了两组原子操作 API 函数，一组是对整形变量进行操作的，一组是对位进行操作的，我们接下来看一下这些 API 函数。

47.2.2 原子整形操作 API 函数

Linux 内核定义了叫做 atomic_t 的结构体来完成整形数据的原子操作，在使用中用原子变量来代替整形变量，此结构体定义在 include/linux/types.h 文件中，定义如下：

示例代码 47.2.2.1 atomic_t 结构体

```
175 typedef struct {
176     int counter;
177 } atomic_t;
```

如果要使用原子操作 API 函数，首先要先定义一个 atomic_t 的变量，如下所示：

```
atomic_t    a;  //定义 a
```

也可以在定义原子变量的时候给原子变量赋初值，如下所示：

```
atomic_t b = ATOMIC_INIT(0);    //定义原子变量 b 并赋初值为 0
```

可以通过宏 ATOMIC_INIT 向原子变量赋初值。

原子变量有了，接下来就是对原子变量进行操作，比如读、写、增加、减少等等，Linux 内核提供了大量的原子操作 API 函数，如表 47.2.2.1 所示：

函数	描述
ATOMIC_INIT(int i)	定义原子变量的时候对其初始化。
int atomic_read(atomic_t *v)	读取 v 的值，并且返回。
void atomic_set(atomic_t *v, int i)	向 v 写入 i 值。
void atomic_add(int i, atomic_t *v)	给 v 加上 i 值。
void atomic_sub(int i, atomic_t *v)	从 v 减去 i 值。
void atomic_inc(atomic_t *v)	给 v 加 1，也就是自增。
void atomic_dec(atomic_t *v)	从 v 减 1，也就是自减
int atomic_dec_return(atomic_t *v)	从 v 减 1，并且返回 v 的值。
int atomic_inc_return(atomic_t *v)	给 v 加 1，并且返回 v 的值。
int atomic_sub_and_test(int i, atomic_t *v)	从 v 减 i，如果结果为 0 就返回真，否则返回假
int atomic_dec_and_test(atomic_t *v)	从 v 减 1，如果结果为 0 就返回真，否则返回假
int atomic_inc_and_test(atomic_t *v)	给 v 加 1，如果结果为 0 就返回真，否则返回假
int atomic_add_negative(int i, atomic_t *v)	给 v 加 i，如果结果为负就返回真，否则返回假

表 47.2.2.1 原子整形操作 API 函数表

如果使用 64 位的 SOC 的话，就要用到 64 位的原子变量，Linux 内核也定义了 64 位原子结构体，如下所示：

示例代码 47.2.2.2 atomic64_t 结构体

```
typedef struct {
    long long counter;
} atomic64_t;
```

相应的也提供了 64 位原子变量的操作 API 函数，这里我们就不详细讲解了，和表 47.2.1.1 中的 API 函数有用法一样，只是将“atomic_”前缀换为“atomic64_”，将 int 换为 long long。如果使用的是 64 位的 SOC，那么就要使用 64 位的原子操作函数。Cortex-A7 是 32 位的架构，所以本书中只使用表 47.2.2.1 中的 32 位原子操作函数。原子变量和相应的 API 函数使用起来很简单，参考如下示例：

示例代码 47.2.2.2 原子变量和 API 函数使用

```
atomic_t v = ATOMIC_INIT(0);    /* 定义并初始化原子变零 v=0  */

atomic_set(10);                 /* 设置 v=10               */
atomic_read(&v);                /* 读取 v 的值，肯定是 10   */
atomic_inc(&v);                 /* v 的值加 1，v=11         */
```

47.2.3 原子位操作 API 函数

位操作也是很常用的操作，Linux 内核也提供了一系列的原子位操作 API 函数，只不过原子位操作不像原子整形变量那样有个 atomic_t 的数据结构，原子位操作是直接对内存进行操作，API 函数如表 47.2.3.1 所示：

函数	描述
void set_bit(int nr, void *p)	将 p 地址的第 nr 位置 1。
void clear_bit(int nr,void *p)	将 p 地址的第 nr 位清零。

void change_bit(int nr, void *p)	将 p 地址的第 nr 位进行翻转。
int test_bit(int nr, void *p)	获取 p 地址的第 nr 位的值。
int test_and_set_bit(int nr, void *p)	将 p 地址的第 nr 位置 1，并且返回 nr 位原来的值。
int test_and_clear_bit(int nr, void *p)	将 p 地址的第 nr 位清零，并且返回 nr 位原来的值。
int test_and_change_bit(int nr, void *p)	将 p 地址的第 nr 位翻转，并且返回 nr 位原来的值。

表 47.2.3.1 原子位操作函数表

47.3 自旋锁

47.3.1 自旋锁简介

原子操作只能对整形变量或者位进行保护，但是，在实际的使用环境中怎么可能只有整形变量或位这么简单的临界区。举个最简单的例子，设备结构体变量就不是整型变量，我们对于结构体中成员变量的操作也要保证原子性，在线程 A 对结构体变量使用期间，应该禁止其他的线程来访问此结构体变量，这些工作原子操作都不能胜任，需要本节要讲的锁机制，在 Linux 内核中就是自旋锁。

当一个线程要访问某个共享资源的时候首先要先获取相应的锁，锁只能被一个线程持有，只要此线程不释放持有的锁，那么其他的线程就不能获取此锁。对于自旋锁而言，如果自旋锁正在被线程 A 持有，线程 B 想要获取自旋锁，那么线程 B 就会处于忙循环-旋转-等待状态，线程 B 不会进入休眠状态或者说去做其他的处理，而是会一直傻傻的在那里“转圈圈”的等待锁可用。比如现在有个公用电话亭，一次肯定只能进去一个人打电话，现在电话亭里面有人正在打电话，相当于获得了自旋锁。此时你到了电话亭门口，因为里面有人，所以你不能进去打电话，相当于没有获取自旋锁，这个时候你肯定是站在原地等待，你可能因为无聊的等待而转圈圈消遣时光，反正就是那里也不能去，要一直等到里面的人打完电话出来。终于，里面的人打完电话出来了，相当于释放了自旋锁，这个时候你就可以使用电话亭打电话了，相当于获取到了自旋锁。

自旋锁的“自旋”也就是“原地打转”的意思，“原地打转”的目的是为了等待自旋锁可以用，可以访问共享资源。把自旋锁比作一个变量 a，变量 a=1 的时候表示共享资源可用，当 a=0 的时候表示共享资源不可用。现在线程 A 要访问共享资源，发现 a=0(自旋锁被其他线程持有)，那么线程 A 就会不断的查询 a 的值，直到 a=1。从这里我们可以看到自旋锁的一个缺点：那就等待自旋锁的线程会一直处于自选状态，这样会浪费处理器时间，降低系统性能，所以自旋锁的持有时间不能太长。所以自旋锁适用于短时期的轻量级加锁，如果遇到需要长时间持有锁的场景那就需要换其他的方法了，这个我们后面会讲解。

Linux 内核使用结构体 spinlock_t 表示自旋锁，结构体定义如下所示：

示例代码 47.3.1.1 spinlock_t 结构体

```
64 typedef struct spinlock {
65     union {
66         struct raw_spinlock rlock;
67 
68 #ifdef CONFIG_DEBUG_LOCK_ALLOC
69 # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
70         struct {
71             u8 __padding[LOCK_PADSIZE];
72             struct lockdep_map dep_map;
```

```
73          };
74 #endif
75      };
76 } spinlock_t;
```

在使用自旋锁之前，肯定要先定义一个自旋锁变量，定义方法如下所示：

```
spinlock_t    lock;   //定义自旋锁
```

定义好自旋锁变量以后就可以使用相应的 API 函数来操作自旋锁。

47.3.2 自旋锁 API 函数

最基本的自旋锁 API 函数如表 47.3.2.1 所示：

函数	描述
DEFINE_SPINLOCK(spinlock_t lock)	定义并初始化一个自选变量。
int spin_lock_init(spinlock_t *lock)	初始化自旋锁。
void spin_lock(spinlock_t *lock)	获取指定的自旋锁，也叫做加锁。
void spin_unlock(spinlock_t *lock)	释放指定的自旋锁。
int spin_trylock(spinlock_t *lock)	尝试获取指定的自旋锁，如果没有获取到就返回 0
int spin_is_locked(spinlock_t *lock)	检查指定的自旋锁是否被获取，如果没有被获取就返回非 0，否则返回 0。

表 47.3.2.1 自旋锁基本 API 函数表

表 47.3.2.1 中的自旋锁 API 函数适用于 SMP 或支持抢占的单 CPU 下线程之间的并发访问，也就是用于线程与线程之间，被自旋锁保护的临界区一定不能调用任何能够引起睡眠和阻塞的 API 函数，否则的话会可能会导致死锁现象的发生。自旋锁会自动禁止抢占，也就说当线程 A 得到锁以后会暂时禁止内核抢占。如果线程 A 在持有锁期间进入了休眠状态，那么线程 A 会自动放弃 CPU 使用权。线程 B 开始运行，线程 B 也想要获取锁，但是此时锁被 A 线程持有，而且内核抢占还被禁止了！线程 B 无法被调度出去，那么线程 A 就无法运行，锁也就无法释放，好了，死锁发生了！

表 47.3.2.1 中的 API 函数用于线程之间的并发访问，如果此时中断也要插一脚，中断也想访问共享资源，那该怎么办呢？首先可以肯定的是，中断里面使用自旋锁，但是在中断里面使用自旋锁的时候，在获取锁之前一定要先禁止本地中断(也就是本 CPU 中断，对于多核 SOC 来说会有多个 CPU 核)，否则可能导致锁死现象的发生，如图 47.3.2.1 所示：

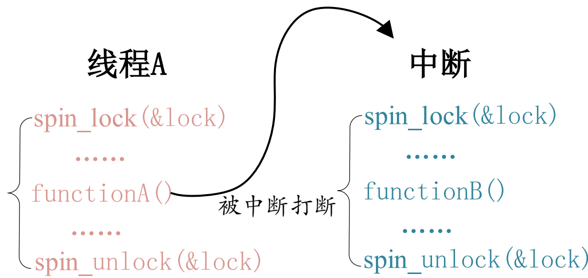


图 47.3.2.1 中断打断线程

在图 47.3.2.1 中，线程 A 先运行，并且获取到了 lock 这个锁，当线程 A 运行 functionA 函数的时候中断发生了，中断抢走了 CPU 使用权。右边的中断服务函数也要获取 lock 这个锁，但是这个锁被线程 A 占有着，中断就会一直自选，等待锁有效。但是在中断服务函数执行完之前，线程 A 是不可能执行的，线程 A 说“你先放手”，中断说“你先放手”，场面就这么僵持着，死锁发生！

最好的解决方法就是获取锁之前关闭本地中断，Linux 内核提供了相应的 API 函数，如表 47.3.2.2 所示：

函数	描述
void spin_lock_irq(spinlock_t *lock)	禁止本地中断，并获取自旋锁。
void spin_unlock_irq(spinlock_t *lock)	激活本地中断，并释放自旋锁。
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)	保存中断状态，禁止本地中断，并获取自旋锁。
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)	将中断状态恢复到以前的状态，并且激活本地中断，释放自旋锁。

表 47.3.2.2 线程与中断并发访问处理 API 函数

使用 spin_lock_irq/spin_unlock_irq 的时候需要用户能够确定加锁之前的中断状态，但实际上内核很庞大，运行也是“千变万化”，我们是很难确定某个时刻的中断状态，因此不推荐使用 spin_lock_irq/spin_unlock_irq。建议使用 spin_lock_irqsave/ spin_unlock_irqrestore，因为这一组函数会保存中断状态，在释放锁的时候会恢复中断状态。一般在线程中使用 spin_lock_irqsave/ spin_unlock_irqrestore，在中断中使用 spin_lock/spin_unlock，示例代码如下所示：

示例代码 47.3.2.1 自旋锁使用示例

```
1  DEFINE_SPINLOCK(lock)                        /* 定义并初始化一个锁    */
2  
3  /* 线程 A */
4  void functionA (){
5      unsigned long flags;                     /* 中断状态              */
6      spin_lock_irqsave(&lock, flags)          /* 获取锁                */
7      /* 临界区 */
8      spin_unlock_irqrestore(&lock, flags)     /* 释放锁                */
9  }
10 
11 /* 中断服务函数 */
12 void irq() {
13     spin_lock(&lock)                         /* 获取锁                */
14     /* 临界区 */
15     spin_unlock(&lock)                       /* 释放锁                */
16 }
```

下半部(BH)也会竞争共享资源，有些资料也会将下半部叫做底半部。关于下半部后面的章节会讲解，如果要在下半部里面使用自旋锁，可以使用表 47.3.2.3 中的 API 函数：

函数	描述
void spin_lock_bh(spinlock_t *lock)	关闭下半部，并获取自旋锁。
void spin_unlock_bh(spinlock_t *lock)	打开下半部，并释放自旋锁。

表 47.3.2.3 下半部竞争处理函数

47.3.3 其他类型的锁

在自旋锁的基础上还衍生出了其他特定场合使用的锁，这些锁在驱动中其实用的不多，更多的是在 Linux 内核中使用，本节我们简单来了解一下这些衍生出来的锁。

1、读写自旋锁

现在有个学生信息表，此表存放着学生的年龄、家庭住址、班级等信息，此表可以随时被修改和读取。此表肯定是数据，那么必须要对其进行保护，如果我们现在使用自旋锁对其进行保护。每次只能一个读操作或者写操作，但是，实际上此表是可以并发读取的。只需要保证在修改此表的时候没人读取，或者在其他人读取此表的时候没有人修改此表就行了。也就是此表的读和写不能同时进行，但是可以多人并发的读取此表。像这样，当某个数据结构符合读/写或生产者/消费者模型的时候就可以使用读写自旋锁。

读写自旋锁为读和写操作提供了不同的锁，一次只能允许一个写操作，也就是只能一个线程持有写锁，而且不能进行读操作。但是当没有写操作的时候允许一个或多个线程持有读锁，可以进行并发的读操作。Linux 内核使用 rwlock_t 结构体表示读写锁，结构体定义如下(删除了条件编译)：

示例代码 47.3.3.1 rwlock_t 结构体

```
typedef struct {
    arch_rwlock_t raw_lock;
} rwlock_t;
```

读写锁操作 API 函数分为两部分，一个是给读使用的，一个是给写使用的，这些 API 函数如表 47.3.3.1 所示：

函数	描述
DEFINE_RWLOCK(rwlock_t lock)	定义并初始化读写锁
void rwlock_init(rwlock_t *lock)	初始化读写锁。
读锁	
void read_lock(rwlock_t *lock)	获取读锁。
void read_unlock(rwlock_t *lock)	释放读锁。
void read_lock_irq(rwlock_t *lock)	禁止本地中断，并且获取读锁。
void read_unlock_irq(rwlock_t *lock)	打开本地中断，并且释放读锁。
void read_lock_irqsave(rwlock_t *lock, unsigned long flags)	保存中断状态，禁止本地中断，并获取读锁。
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags)	将中断状态恢复到以前的状态，并且激活本地中断，释放读锁。
void read_lock_bh(rwlock_t *lock)	关闭下半部，并获取读锁。
void read_unlock_bh(rwlock_t *lock)	打开下半部，并释放读锁。
写锁	
void write_lock(rwlock_t *lock)	获取读锁。
void write_unlock(rwlock_t *lock)	释放读锁。
void write_lock_irq(rwlock_t *lock)	禁止本地中断，并且获取读锁。
void write_unlock_irq(rwlock_t *lock)	打开本地中断，并且释放读锁。
void write_lock_irqsave(rwlock_t *lock, unsigned long flags)	保存中断状态，禁止本地中断，并获取读锁。
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags)	将中断状态恢复到以前的状态，并且激活本地中断，释放读锁。
void write_lock_bh(rwlock_t *lock)	关闭下半部，并获取读锁。
void write_unlock_bh(rwlock_t *lock)	打开下半部，并释放读锁。

表 47.3.3.1 读写锁 API 函数

2、顺序锁

顺序锁在读写锁的基础上衍生而来的，使用读写锁的时候读操作和写操作不能同时进行。使用顺序锁的话可以允许在写的时候进行读操作，也就是实现同时读写，但是不允许同时进行并发的写操作。虽然顺序锁的读和写操作可以同时进行，但是如果在读的过程中发生了写操作，最好重新进行读取，保证数据完整性。顺序锁保护的资源不能是指针，因为如果在写操作的时候可能会导致指针无效，而这个时候恰巧有读操作访问指针的话就可能导致意外发生，比如读取野指针导致系统崩溃。Linux 内核使用 seqlock_t 结构体表示顺序锁，结构体定义如下：

示例代码 47.3.3.2 seqlock_t 结构体

```
typedef struct {
    struct seqcount seqcount;
    spinlock_t lock;
} seqlock_t;
```

关于顺序锁的 API 函数如表 47.3.3.2 所示：

函数	描述
DEFINE_SEQLOCK(seqlock_t sl)	定义并初始化顺序锁
void seqlock_ini seqlock_t *sl)	初始化顺序锁。
顺序锁写操作	
void write_seqlock(seqlock_t *sl)	获取写顺序锁。
void write_sequnlock(seqlock_t *sl)	释放写顺序锁。
void write_seqlock_irq(seqlock_t *sl)	禁止本地中断，并且获取写顺序锁
void write_sequnlock_irq(seqlock_t *sl)	打开本地中断，并且释放写顺序锁。
void write_seqlock_irqsave(seqlock_t *sl, unsigned long flags)	保存中断状态，禁止本地中断，并获取写顺序锁。
void write_sequnlock_irqrestore(seqlock_t *sl, unsigned long flags)	将中断状态恢复到以前的状态，并且激活本地中断，释放写顺序锁。
void write_seqlock_bh(seqlock_t *sl)	关闭下半部，并获取写读锁。
void write_sequnlock_bh(seqlock_t *sl)	打开下半部，并释放写读锁。
顺序锁读操作	
unsigned read_seqbegin(const seqlock_t *sl)	读单元访问共享资源的时候调用此函数，此函数会返回顺序锁的顺序号。
unsigned read_seqretry(const seqlock_t *sl, unsigned start)	读结束以后调用此函数检查在读的过程中有没有对资源进行写操作，如果有的话就要重读

表 47.3.3.2 顺序锁 API 函数表

47.3.4 自旋锁使用注意事项

综合前面关于自旋锁的信息，我们需要在使用自旋锁的时候要注意一下几点：

①、因为在等待自旋锁的时候处于“自旋”状态，因此锁的持有时间不能太长，一定要短，否则的话会降低系统性能。如果临界区比较大，运行时间比较长的话要选择其他的并发处理方式，比如稍后要讲的信号量和互斥体。

②、自旋锁保护的临界区内不能调用任何可能导致线程休眠的 API 函数，否则的话可能导致死锁。

③、不能递归申请自旋锁，因为一旦通过递归的方式申请一个你正在持有的锁，那么你就必须“自旋”，等待锁被释放，然而你正处于“自旋”状态，根本没法释放锁。结果就是自己把自己锁死了！

④、在编写驱动程序的时候我们必须考虑到驱动的可移植性，因此不管你用的是单核的还是多核的 SOC，都将其当做多核 SOC 来编写驱动程序。

47.4 信号量

47.4.1 信号量简介

大家如果有学习过 FreeRTOS 或者 UCOS 的话就应该对信号量很熟悉，因为信号量是同步的一种方式。Linux 内核也提供了信号量机制，信号量常常用于控制对共享资源的访问。举一个很常见的例子，某个停车场有 100 个停车位，这 100 个停车位大家都可以用，对于大家来说这 100 个停车位就是共享资源。假设现在这个停车场正常运行，你要把车停到这个这个停车场肯定要先看一下现在停了多少车了？还有没有停车位？当前停车数量就是一个信号量，具体的停车数量就是这个信号量值，当这个值到 100 的时候说明停车场满了。停车场满的时你可以等一会看看有没有其他的车开出停车场，当有车开出停车场的时候停车数量就会减一，也就是说信号量减一，此时你就可以把车停进去了，你把车停进去以后停车数量就会加一，也就是信号量加一。这就是一个典型的使用信号量进行共享资源管理的案例，在这个案例中使用的就是计数型信号量。

相比于自旋锁，信号量可以使线程进入休眠状态，比如 A 与 B、C 合租了一套房子，这个房子只有一个厕所，一次只能一个人使用。某一天早上 A 去上厕所了，过了一会 B 也想用厕所，因为 A 在厕所里面，所以 B 只能等到 A 用来了才能进去。B 要么就一直在厕所门口等着，等 A 出来，这个时候就相当于自旋锁。B 也可以告诉 A，让 A 出来以后通知他一下，然后 B 继续回房间睡觉，这个时候相当于信号量。可以看出，使用信号量会提高处理器的使用效率，毕竟不用一直傻乎乎的在那里“自选”等待。但是，信号量的开销要比自旋锁大，因为信号量使线程进入休眠状态以后会切换线程，切换线程就会有开销。总结一下信号量的特点：

①、因为信号量可以使等待资源线程进入休眠状态，因此适用于那些占用资源比较久的场合。

②、因此信号量不能用于中断中，因为信号量会引起休眠，中断不能休眠。

③、如果共享资源的持有时间比较短，那就不适合使用信号量了，因为频繁的休眠、切换线程引起的开销要远大于信号量带来的那点优势。

信号量有一个信号量值，相当于一个房子有 10 把钥匙，这 10 把钥匙就相当于信号量值为 10。因此，可以通过信号量来控制访问共享资源的访问数量，如果要想进房间，那就要先获取一把钥匙，信号量值减 1，直到 10 把钥匙都被拿走，信号量值为 0，这个时候就不允许任何人进入房间了，因为没钥匙了。如果有人从房间出来，那他要归还他所持有的那把钥匙，信号量值加 1，此时有 1 把钥匙了，那么可以允许进去一个人。相当于通过信号量控制访问资源的线程数，在初始化的时候将信号量值设置的大于 1，那么这个信号量就是计数型信号量，计数型信号量不能用于互斥访问，因为它允许多个线程同时访问共享资源。如果要互斥的访问共享资源那么信号量的值就不能大于 1，此时的信号量就是一个二值信号量。

47.4.2 信号量 API 函数

Linux 内核使用 semaphore 结构体表示信号量，结构体内容如下所示：

示例代码 47.4.2.1 semaphore 结构体

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

要想使用信号量就得先定义，然后初始化信号量。有关信号量的 API 函数如表 47.4.2.1 所示：

函数	描述
DEFINE_SEAMPHORE(name)	定义一个信号量，并且设置信号量的值为 1。
void sema_init(struct semaphore *sem, int val)	初始化信号量 sem，设置信号量值为 val。
void down(struct semaphore *sem)	获取信号量，因为会导致休眠，因此不能在中断中使用。
int down_trylock(struct semaphore *sem);	尝试获取信号量，如果能获取到信号量就获取，并且返回 0。如果不能就返回非 0，并且不会进入休眠。
int down_interruptible(struct semaphore *sem)	获取信号量，和 down 类似，只是使用 down 进入休眠状态的线程不能被信号打断。而使用此函数进入休眠以后是可以被信号打断的。
void up(struct semaphore *sem)	释放信号量

表 47.4.2.1 信号量 API 函数

信号量的使用如下所示：

示例代码 47.4.2.2 信号量使用示例

```
struct semaphore sem;    /* 定义信号量   */

sema_init(&sem, 1);      /* 初始化信号量 */

down(&sem);              /* 申请信号量   */
/* 临界区 */
up(&sem);                /* 释放信号量   */
```

47.5 互斥体

47.5.1 互斥体简介

在 FreeRTOS 和 UCOS 中也有互斥体，将信号量的值设置为 1 就可以使用信号量进行互斥访问了，虽然可以通过信号量实现互斥，但是 Linux 提供了一个比信号量更专业的机制来进行互斥，它就是互斥体—mutex。互斥访问表示一次只有一个线程可以访问共享资源，不能递归申请互斥体。在我们编写 Linux 驱动的时候遇到需要互斥访问的地方建议使用 mutex。Linux 内核使用 mutex 结构体表示互斥体，定义如下(省略条件编译部分)：

示例代码 47.5.1.1 mutex 结构体

```
struct mutex {
    /* 1: unlocked, 0: locked, negative: locked, possible waiters */
    atomic_t        count;
    spinlock_t      wait_lock;
```

```
};
```

在使用 mutex 之前要先定义一个 mutex 变量。在使用 mutex 的时候要注意如下几点：

①、mutex 可以导致休眠，因此不能在中断中使用 mutex，中断中只能使用自旋锁。

②、和信号量一样，mutex 保护的临界区可以调用引起阻塞的 API 函数。

③、因为一次只有一个线程可以持有 mutex，因此，必须由 mutex 的持有者释放 mutex。并且 mutex 不能递归上锁和解锁。

47.5.2 互斥体 API 函数

有关互斥体的 API 函数如表 47.5.2.1 所示：

函数	描述
DEFINE_MUTEX(name)	定义并初始化一个 mutex 变量。
void mutex_init(mutex *lock)	初始化 mutex。
void mutex_lock(struct mutex *lock)	获取 mutex，也就是给 mutex 上锁。如果获取不到就进休眠。
void mutex_unlock(struct mutex *lock)	释放 mutex，也就给 mutex 解锁。
int mutex_trylock(struct mutex *lock)	尝试获取 mutex，如果成功就返回 1，如果失败就返回 0。
int mutex_is_locked(struct mutex *lock)	判断 mutex 是否被获取，如果是的话就返回 1，否则返回 0。
int mutex_lock_interruptible(struct mutex *lock)	使用此函数获取信号量失败进入休眠以后可以被信号打断。

表 47.5.2.1 互斥体 API 函数

互斥体的使用如下所示：

示例代码 47.5.2.1 互斥体使用示例

```
1 struct mutex lock;    /* 定义一个互斥体      */
2 mutex_init(&lock);    /* 初始化互斥体        */
3
4 mutex_lock(&lock);    /* 上锁               */
5 /* 临界区 */
6 mutex_unlock(&lock);   /* 解锁                */
```

关于 Linux 中的并发和竞争就讲解到这里，Linux 内核还有很多其他的处理并发和竞争的机制，本章我们主要讲解了常用的原子操作、自旋锁、信号量和互斥体。以后我们在编写 Linux 驱动的时候就会频繁的使用到这几种机制，希望大家能够深入理解这几个常用的机制。

第四十八章 Linux 并发与竞争实验

在上一章中我们学习了 Linux 下的并发与竞争，并且学习了四种常用的处理并发和竞争的机制：原子操作、自旋锁、信号量和互斥体。本章我们就通过四个实验来学习如何在驱动中使用这四种机制。

48.1 原子操作实验

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 7_atomic。

本例程我们在第四十五章的 gpioled.c 文件基础上完成。在本节使用中我们使用原子操作来实现对 LED 这个设备的互斥访问，也就是一次只允许一个应用程序可以使用 LED 灯。

48.1.1 实验程序编写

1、修改设备树文件

因为本章实验是在第四十五章实验的基础上完成的，因此不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第四十五章实验驱动文件 gpioled.c 的基础上修改而来。新建名为“7_atomic”的文件夹，然后在 7_atomic 文件夹里面创建 vscode 工程，工作区命名为“atomic”。将 5_gpioled 实验中的 gpioled.c 复制到 7_atomic 文件夹中，并且重命名为 atomic.c。本节实验重点就是使用 atomic 来实现一次只能允许一个应用访问 LED，所以我们只需要在 atomic.c 文件源码的基础上加上添加 atomic 相关代码即可，完成以后的 atomic.c 文件内容如下所示：

示例代码 48.1.1.1 atomic.c 文件代码段

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
17 /***************************************************************
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名      : atomic.c
20 作者        : 左忠凯
21 版本        : V1.0
22 描述        : 原子操作实验，使用原子变量来实现对实现设备的互斥访问
23 其他        : 无
24 论坛        : www.openedv.com
25 日志        : 初版 V1.0 2019/7/18 左忠凯创建
26 ***************************************************************/
```

```
27 #define GPIOLED_CNT       1          /* 设备号个数   */
28 #define GPIOLED_NAME      "gpioled"  /* 名字         */
29 #define LEDOFF            0          /* 关灯         */
30 #define LEDON             1          /* 开灯         */
31 
32 /* gpioled 设备结构体 */
33 struct gpioled_dev{
34     dev_t devid;               /* 设备号                     */
35     struct cdev cdev;          /* cdev                       */
36     struct class *class;       /* 类                         */
37     struct device *device;     /* 设备                       */
38     int major;                 /* 主设备号                   */
39     int minor;                 /* 次设备号                   */
40     struct device_node  *nd;  /* 设备节点                   */
41     int led_gpio;              /* led 所使用的 GPIO 编号    */
42     atomic_t lock;             /* 原子变量                   */
43 };
44 
45 struct gpioled_dev gpioled; /* led 设备                     */
46 
47 /*
48  * @description    : 打开设备
49  * @param – inode : 传递给驱动的 inode
50  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
51  *                 一般在 open 的时候将 private_data 指向设备结构体。
52  * @return         : 0 成功;其他 失败
53  */
54 static int led_open(struct inode *inode, struct file *filp)
55 {
56     /* 通过判断原子变量的值来检查 LED 有没有被别的应用使用 */
57     if (!atomic_dec_and_test(&gpioled.lock)) {
58         atomic_inc(&gpioled.lock);/* 小于 0 的话就加 1,使其原子变量等于 0 */
59         return -EBUSY;            /* LED 被使用，返回忙 */
60     }
61 
62     filp->private_data = &gpioled; /* 设置私有数据 */
63     return 0;
64 }
65 
66 /*
67  * @description    : 从设备读取数据
68  * @param - filp  : 要打开的设备文件(文件描述符)
69  * @param - buf   : 返回给用户空间的数据缓冲区
```

```
70   * @param - cnt   : 要读取的数据长度
71   * @param - offt  : 相对于文件首地址的偏移
72   * @return        : 读取的字节数，如果为负值，表示读取失败
73   */
74  static ssize_t led_read(struct file *filp, char __user *buf,
                           size_t cnt, loff_t *offt)
75  {
76      return 0;
77  }
78
79  /*
80   * @description   : 向设备写数据
81   * @param - filp  : 设备文件，表示打开的文件描述符
82   * @param - buf   : 要写给设备写入的数据
83   * @param - cnt   : 要写入的数据长度
84   * @param - offt  : 相对于文件首地址的偏移
85   * @return        : 写入的字节数，如果为负值，表示写入失败
86   */
87  static ssize_t led_write(struct file *filp, const char __user *buf,
                            size_t cnt, loff_t *offt)
88  {
89      int retvalue;
90      unsigned char databuf[1];
91      unsigned char ledstat;
92      struct gpioled_dev *dev = filp->private_data;
93
94      retvalue = copy_from_user(databuf, buf, cnt);
95      if(retvalue < 0) {
96          printk("kernel write failed!\r\n");
97          return -EFAULT;
98      }
99
100     ledstat = databuf[0];            /* 获取状态值   */
101
102     if(ledstat == LEDON) {
103         gpio_set_value(dev->led_gpio, 0);  /* 打开 LED 灯  */
104     } else if(ledstat == LEDOFF) {
105         gpio_set_value(dev->led_gpio, 1);  /* 关闭 LED 灯  */
106     }
107     return 0;
108 }
109
110 /*
```

```
111  * @description   : 关闭/释放设备
112  * @param - filp  : 要关闭的设备文件(文件描述符)
113  * @return        : 0 成功;其他 失败
114  */
115 static int led_release(struct inode *inode, struct file *filp)
116 {
117     struct gpioled_dev *dev = filp->private_data;
118 
119     /* 关闭驱动文件的时候释放原子变量 */
120     atomic_inc(&dev->lock);
121     return 0;
122 }
123 
124 /* 设备操作函数 */
125 static struct file_operations gpioled_fops = {
126     .owner = THIS_MODULE,
127     .open = led_open,
128     .read = led_read,
129     .write = led_write,
130     .release =  led_release,
131 };
132 
133 /*
134  * @description   : 驱动入口函数
135  * @param         : 无
136  * @return        : 无
137  */
138 static int __init led_init(void)
139 {
140     int ret = 0;
141 
142     /* 初始化原子变量 */
143     atomic_set(&gpioled.lock, 1);   /* 原子变量初始值为 1 */
144 
145     /* 设置 LED 所使用的 GPIO */
146     /* 1、获取设备节点：gpioled */
147     gpioled.nd = of_find_node_by_path("/gpioled");
148     if(gpioled.nd == NULL) {
149         printk("gpioled node not find!\r\n");
150         return -EINVAL;
151     } else {
152         printk("gpioled node find!\r\n");
153     }
```

```
154
155     /* 2、 获取设备树中的 gpio 属性，得到 LED 所使用的 LED 编号 */
156     gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
157     if(gpioled.led_gpio < 0) {
158         printk("can't get led-gpio");
159         return -EINVAL;
160     }
161     printk("led-gpio num = %d\r\n", gpioled.led_gpio);
162
163     /* 3、设置 GPIO1_IO03 为输出，并且输出高电平，默认关闭 LED 灯 */
164     ret = gpio_direction_output(gpioled.led_gpio, 1);
165     if(ret < 0) {
166         printk("can't set gpio!\r\n");
167     }
168
169     /* 注册字符设备驱动 */
170     /* 1、创建设备号 */
171     if (gpioled.major) {                        /*  定义了设备号 */
172         gpioled.devid = MKDEV(gpioled.major, 0);
173         register_chrdev_region(gpioled.devid, GPIOLED_CNT,
                                    GPIOLED_NAME);
174     } else {                                    /* 没有定义设备号 */
175         alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,
                                   GPIOLED_NAME);  /* 申请设备号 */
176         gpioled.major = MAJOR(gpioled.devid);/* 获取分配号的主设备号 */
177         gpioled.minor = MINOR(gpioled.devid);/* 获取分配号的次设备号 */
178     }
179     printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
                gpioled.minor);
180
181     /* 2、初始化 cdev */
182     gpioled.cdev.owner = THIS_MODULE;
183     cdev_init(&gpioled.cdev, &gpioled_fops);
184
185     /* 3、添加一个 cdev */
186     cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
187
188     /* 4、创建类 */
189     gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
190     if (IS_ERR(gpioled.class)) {
191         return PTR_ERR(gpioled.class);
192     }
193
```

```
194     /* 5、创建设备 */
195     gpioled.device = device_create(gpioled.class, NULL,
                                gpioled.devid, NULL, GPIOLED_NAME);
196     if (IS_ERR(gpioled.device)) {
197         return PTR_ERR(gpioled.device);
198     }
199 
200     return 0;
201 }
202 
203 /*
204  * @description   : 驱动出口函数
205  * @param         : 无
206  * @return        : 无
207  */
208 static void __exit led_exit(void)
209 {
210     /* 注销字符设备驱动 */
211     cdev_del(&gpioled.cdev);/*  删除 cdev */
212     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
213 
214     device_destroy(gpioled.class, gpioled.devid);
215     class_destroy(gpioled.class);
216 }
217 
218 module_init(led_init);
219 module_exit(led_exit);
220 MODULE_LICENSE("GPL");
221 MODULE_AUTHOR("zuozhongkai");
```

第 42 行，原子变量 lock，用来实现一次只能允许一个应用访问 LED 灯，led_init 驱动入口函数会将 lock 的值设置为 1。

第 57~60 行，每次调用 open 函数打开驱动设备的时候先申请 lock，如果申请成功的话就表示 LED 灯还没有被其他的应用使用，如果申请失败就表示 LED 灯正在被其他的应用程序使用。每次打开驱动设备的时候先使用 atomic_dec_and_test 函数将 lock 减 1，如果 atomic_dec_and_test 函数返回值为真就表示 lock 当前值为 0，说明设备可以使用。如果 atomic_dec_and_test 函数返回值为假，就表示 lock 当前值为负数(lock 值默认是 1)，lock 值为负数的可能性只有一个，那就是其他设备正在使用 LED。其他设备正在使用 LED 灯，那么就只能退出了，在退出之前调用函数 atomic_inc 将 lock 加 1，因为此时 lock 的值被减成了负数，必须要对其加 1，将 lock 的值变为 0。

第 120 行，LED 灯使用完毕，应用程序调用 close 函数关闭的驱动文件，led_release 函数执行，调用 atomic_inc 释放 lcok，也就是将 lock 加 1。

第 143 行，初始化原子变量 lock，初始值设置为 1，这样每次就只允许一个应用使用 LED 灯。

3、编写测试 APP

新建名为 atomicApp.c 的测试 APP，在里面输入如下所示内容：

示例代码 48.1.1.2 atomicApp.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名      : atomicApp.c
11 作者        : 左忠凯
12 版本        : V1.0
13 描述        : 原子变量测试 APP，测试原子变量能不能实现一次
14               只允许一个应用程序使用 LED。
15 其他        : 无
16 使用方法    : ./atomicApp /dev/gpioled  0 关闭 LED 灯
17               ./atomicApp /dev/gpioled  1 打开 LED 灯
18 论坛        : www.openedv.com
19 日志        : 初版 V1.0 2019/1/30 左忠凯创建
20 ***************************************************************/
21
22 #define LEDOFF    0
23 #define LEDON     1
24
25 /*
26  * @description    : main 主程序
27  * @param - argc   : argv 数组元素个数
28  * @param - argv   : 具体参数
29  * @return         : 0 成功;其他 失败
30  */
31 int main(int argc, char *argv[])
32 {
33     int fd, retvalue;
34     char *filename;
35     unsigned char cnt = 0;
36     unsigned char databuf[1];
37
38     if(argc != 3){
39         printf("Error Usage!\r\n");
40         return -1;
```

```
41      }
42
43      filename = argv[1];
44
45      /* 打开 beep 驱动 */
46      fd = open(filename, O_RDWR);
47      if(fd < 0){
48          printf("file %s open failed!\r\n", argv[1]);
49          return -1;
50      }
51
52      databuf[0] = atoi(argv[2]); /* 要执行的操作：打开或关闭 */
53
54      /* 向/dev/gpioled 文件写入数据 */
55      retvalue = write(fd, databuf, sizeof(databuf));
56      if(retvalue < 0){
57          printf("LED Control Failed!\r\n");
58          close(fd);
59          return -1;
60      }
61
62      /* 模拟占用 25S LED */
63      while(1) {
64          sleep(5);
65          cnt++;
66          printf("App running times:%d\r\n", cnt);
67          if(cnt >= 5) break;
68      }
69
70      printf("App running finished!");
71      retvalue = close(fd); /* 关闭文件 */
72      if(retvalue < 0){
73          printf("file %s close failed!\r\n", argv[1]);
74          return -1;
75      }
76      return 0;
77  }
```

atomicApp.c 中的内容就是在第四十五章的 ledAPP.c 的基础上修改而来的，重点是加入了第 63~68 行的模拟占用 25 秒 LED 的代码。测试 APP 在获取到 LED 灯驱动的使用权以后会使用 25S，在使用的这段时间如果有其他的应用也去获取 LED 灯使用权的话肯定会失败！

48.1.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 atomic.o，Makefile 内容如下所示：

示例代码 48.1.2.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := atomic.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 atomic.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“atomic.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 atomicApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc atomicApp.c -o atomicApp
```

编译成功以后就会生成 atomicApp 这个应用程序。

3、运行测试

将上一小节编译出来的 atomic.ko 和 atomicApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 atomic.ko 驱动模块：

```
depmod                //第一次加载驱动的时候需要运行此命令
modprobe atomic.ko    //加载驱动
```

驱动加载成功以后就可以使用 atomicApp 软件来测试驱动是否工作正常，输入如下命令以后台运行模式打开 LED 灯，“&”表示在后台运行 atomicApp 这个软件：

```
./atomicApp /dev/gpioled 1&       //打开 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否点亮，然后每隔 5 秒都会输出一行“App running times ”，如图 48.1.2.1 所示：

```
/lib/modules/4.1.15 #
/lib/modules/4.1.15 # ./atomicApp /dev/gpioled 1&
/lib/modules/4.1.15 # App running times:1
App running times:2
```

图 48.1.2.1 打开 LED 灯

从图 48.1.2.1 可以看出，atomicApp 运行正常，输出了“App running times:1”和“App running times:2”，这就是模拟 25S 占用，说明 atomicApp 这个软件正在使用 LED 灯。此时再输入如下命令关闭 LED 灯：

```
./atomicApp  /dev/gpioled 0 //关闭 LED 灯
```

输入上述命令以后会发现如图 48.1.2.2 所示输入信息：

```
/lib/modules/4.1.15 # ./atomicApp /dev/gpioled 0
file /dev/gpioled open failed!
```

图 48.1.2.2 关闭 LED 灯

从图 48.1.2.2 可以看出，打开/dev/gpioled 失败！原因是在图 48.1.2.1 中运行的 atomicAPP 软件正在占用/dev/gpioled，如果再次运行 atomicApp 软件去操作/dev/gpioled 肯定会失败。必须等待图 48.1.2.1 中的 atomicApp 运行结束，也就是 25S 结束以后其他软件才能去操作/dev/gpioled。这个就是采用原子变量实现一次只能有一个应用程序访问 LED 灯。

如果要卸载驱动的话输入如下命令即可：

```
rmmod atomic.ko
```

48.2 自旋锁实验

上一节我们使用原子变量实现了一次只能有一个应用程序访问 LED 灯，本节我们使用自旋锁来实现此功能。在使用自旋锁之前，先回顾一下自旋锁的使用注意事项：

①、自旋锁保护的临界区要尽可能的短，因此在 open 函数中申请自旋锁，然后在 release 函数中释放自旋锁的方法就不可取。我们可以使用一个变量来表示设备的使用情况，如果设备被使用了那么变量就加一，设备被释放以后变量就减 1，我们只需要使用自旋锁保护这个变量即可。

②、考虑驱动的兼容性，合理的选择 API 函数。

综上所述，在本节例程中，我们通过定义一个变量 dev_stats 表示设备的使用情况，dev_stats 为 0 的时候表示设备没有被使用，dev_stats 大于 0 的时候表示设备被使用。驱动 open 函数中先判断 dev_stats 是否为 0，也就是判断设备是否可用，如果为 0 的话就使用设备，并且将 dev_stats 加 1，表示设备被使用了。使用完以后在 release 函数中将 dev_stats 减 1，表示设备没有被使用了。因此真正实现设备互斥访问的是变量 dev_stats，但是我们要使用自旋锁对 dev_stats 来做保护。

48.2.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的，同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 atomic.c 的基础上修改而来。新建名为“8_spinlock”的文件夹，然后在 8_spinlock 文件夹里面创建 vscode 工程，工作区命名为“spinlock”。将 7_atomic 实验中的 atomic.c 复制到 8_spinlock 文件夹中，并且重命名为 spinlock.c。将原来使用 atomic 的地方换为 spinlock 即可，其他代码不需要修改，完成以后的 spinlock.c 文件内容如下所示(有省略)：

示例代码 48.2.1.1 spinlock.c 文件代码

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
......
17  /***************************************************************
18  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19  文件名      : spinlock.c
```

```
20 作者          : 左忠凯
21 版本          : V1.0
22 描述          : 自旋锁实验，使用自旋锁来实现对实现设备的互斥访问
23 其他          : 无
24 论坛          : www.openedv.com
25 日志          : 初版 V1.0 2019/7/18 左忠凯创建
26 ***************************************************************/
27 #define GPIOLED_CNT          1          /* 设备号个数   */
28 #define GPIOLED_NAME         "gpioled"  /* 名字         */
29 #define LEDOFF               0          /* 关灯         */
30 #define LEDON                1          /* 开灯         */
31 
32 
33 /* gpioled 设备结构体 */
34 struct gpioled_dev{
35     dev_t devid;                /* 设备号              */
36     struct cdev cdev;           /* cdev                */
37     struct class *class;        /* 类                  */
38     struct device *device;      /* 设备                */
39     int major;                  /* 主设备号            */
40     int minor;                  /* 次设备号            */
41     struct device_node  *nd;    /* 设备节点            */
42     int led_gpio;               /* led 所使用的 GPIO 编号   */
43     int dev_stats;            /* 设备状态，0，设备未使用;>0,设备已经被使用 */
44     spinlock_t lock;            /* 自旋锁              */
45 };
46 
47 struct gpioled_dev gpioled; /* led 设备             */
48 
49 /*
50  * @description   : 打开设备
51  * @param – inode : 传递给驱动的 inode
52  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
53  *                  一般在 open 的时候将 private_data 指向设备结构体。
54  * @return        : 0 成功;其他 失败
55  */
56 static int led_open(struct inode *inode, struct file *filp)
57 {
58     unsigned long flags;
59     filp->private_data = &gpioled;  /* 设置私有数据 */
60 
61     spin_lock_irqsave(&gpioled.lock, flags);  /* 上锁            */
62     if (gpioled.dev_stats) {                  /* 如果设备被使用了  */
```

```
63          spin_unlock_irqrestore(&gpioled.lock, flags); /* 解锁 */
64          return -EBUSY;
65      }
66      gpioled.dev_stats++;    /* 如果设备没有打开，那么就标记已经打开了 */
67      spin_unlock_irqrestore(&gpioled.lock, flags);/* 解锁 */
68  
69      return 0;
70  }
......
116 /*
117  * @description   : 关闭/释放设备
118  * @param - filp  : 要关闭的设备文件(文件描述符)
119  * @return        : 0 成功;其他 失败
120  */
121 static int led_release(struct inode *inode, struct file *filp)
122 {
123     unsigned long flags;
124     struct gpioled_dev *dev = filp->private_data;
125 
126     /* 关闭驱动文件的时候将 dev_stats 减 1 */
127     spin_lock_irqsave(&dev->lock, flags);   /* 上锁 */
128     if (dev->dev_stats) {
129         dev->dev_stats--;
130     }
131     spin_unlock_irqrestore(&dev->lock, flags);/* 解锁 */
132 
133     return 0;
134 }
135 
136 /* 设备操作函数 */
137 static struct file_operations gpioled_fops = {
138     .owner = THIS_MODULE,
139     .open = led_open,
140     .read = led_read,
141     .write = led_write,
142     .release =  led_release,
143 };
144 
145 /*
146  * @description   : 驱动入口函数
147  * @param         : 无
148  * @return        : 无
149  */
```

```
150 static int __init led_init(void)
151 {
152     int ret = 0;
153 
154     /*  初始化自旋锁 */
155     spin_lock_init(&gpioled.lock);
......
212     return 0;
213 }
214 
215 /*
216  * @description   : 驱动出口函数
217  * @param         : 无
218  * @return        : 无
219  */
220 static void __exit led_exit(void)
221 {
222     /* 注销字符设备驱动 */
223     cdev_del(&gpioled.cdev);/*  删除 cdev */
224     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
225 
226     device_destroy(gpioled.class, gpioled.devid);
227     class_destroy(gpioled.class);
228 }
229 
230 module_init(led_init);
231 module_exit(led_exit);
232 MODULE_LICENSE("GPL");
233 MODULE_AUTHOR("zuozhongkai");
```

第 43 行，dev_stats 表示设备状态，如果为 0 的话表示设备还没有被使用，如果大于 0 的话就表示设备已经被使用了。

第 44 行，定义自旋锁变量 lock。

第 61~67 行，使用自旋锁实现对设备的互斥访问，第 61 行调用 spin_lock_irqsave 函数获取锁，为了考虑到驱动兼容性，这里并没有使用 spin_lock 函数来获取锁。第 62 行判断 dev_stats 是否大于 0，如果是的话表示设备已经被使用了，那么就调用 spin_unlock_irqrestore 函数释放锁，并且返回-EBUSY。如果设备没有被使用的话就在第 66 行将 dev_stats 加 1，表示设备要被使用了，然后调用 spin_unlock_irqrestore 函数释放锁。自旋锁的工作就是保护 dev_stats 变量，真正实现对设备互斥访问的是 dev_stats。

第 126~131 行，在 release 函数中将 dev_stats 减 1，表示设备被释放了，可以被其他的应用程序使用。将 dev_stats 减 1 的时候需要自旋锁对其进行保护。

第 155 行，在驱动入口函数 led_init 中调用 spin_lock_init 函数初始化自旋锁。

3、编写测试 APP

测试 APP 使用 48.1.1 小节中的 atomicApp.c 即可，将 7_atomic 中的 atomicApp.c 文件到本例程中，并将 atomicApp.c 重命名为 spinlockApp.c 即可。

48.2.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 spinlock.o，Makefile 内容如下所示：

示例代码 48.2.2.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := spinlock.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 spinlock.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“spinlock.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 spinlockApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc spinlockApp.c -o spinlockApp
```

编译成功以后就会生成 spinlockApp 这个应用程序。

3、运行测试

将上一小节编译出来的 spinlock.ko 和 spinlockApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 spinlock.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe spinlock.ko    //加载驱动
```

驱动加载成功以后就可以使用 spinlockApp 软件测试驱动是否工作正常，测试方法和 48.1.2 小节中一样，先输入如下命令让 spinlockAPP 软件模拟占用 25S 的 LED 灯：

```
./atomicApp /dev/gpioled 1&        //打开 LED 灯
```

紧接着再输入如下命令关闭 LED 灯：

```
./atomicApp   /dev/gpioled 0        //关闭 LED 灯
```

看一下能不能关闭 LED 灯，驱动正常工作的话并不会马上关闭 LED 灯，会提示你“file /dev/gpioled open failed!”，必须等待第一个 atomicApp 软件运行完成(25S 计时结束)才可以再次操作 LED 灯。

如果要卸载驱动的话输入如下命令即可：

```
rmmod spinlock.ko
```

48.3 信号量实验

本节我们来使用信号量实现了一次只能有一个应用程序访问 LED 灯，信号量可以导致休眠，因此信号量保护的临界区没有运行时间限制，可以在驱动的 open 函数申请信号量，然后在 release 函数中释放信号量。但是信号量不能用在中断中，本节实验我们不会在中断中使用信号量。

48.3.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的，同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 spinlock.c 的基础上修改而来。新建名为“9_semaphore”的文件夹，然后在 9_semaphore 文件夹里面创建 vscode 工程，工作区命名为“semaphore”。将 8_spinlock 实验中的 spinlock.c 复制到 9_semaphore 文件夹中，并且重命名为 semaphore.c。将原来使用到自旋锁的地方换为信号量即可，其他的内容基本不变，完成以后的 semaphore.c 文件内容如下所示(有省略)：

示例代码 48.3.1.1 semaphore.c 文件代码

```
1   #include <linux/types.h>
......
14  #include <linux/semaphore.h>
15  #include <asm/mach/map.h>
16  #include <asm/uaccess.h>
17  #include <asm/io.h>
18  /***************************************************************
19  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20  文件名      : semaphore.c
21  作者        : 左忠凯
22  版本        : V1.0
23  描述        : 信号量实验，使用信号量来实现对实现设备的互斥访问
24  其他        : 无
25  论坛        : www.openedv.com
26  日志        : 初版 V1.0 2019/7/18 左忠凯创建
27  ***************************************************************/
28  #define GPIOLED_CNT      1          /* 设备号个数   */
29  #define GPIOLED_NAME     "gpioled"  /* 名字         */
30  #define LEDOFF           0          /* 关灯         */
31  #define LEDON            1          /* 开灯         */
32
33  /* gpioled 设备结构体 */
34  struct gpioled_dev{
35      dev_t devid;            /* 设备号           */
36      struct cdev cdev;       /* cdev             */
```

```
37      struct class *class;     /* 类                      */
38      struct device *device;   /* 设备                    */
39      int major;               /* 主设备号                */
40      int minor;               /* 次设备号                */
41      struct device_node  *nd; /* 设备节点                */
42      int led_gpio;            /* led 所使用的 GPIO 编号  */
43      struct semaphore sem;    /* 信号量                  */
44  };
45
46  struct gpioled_dev gpioled; /* led 设备 */
47
48  /*
49   * @description   : 打开设备
50   * @param – inode : 传递给驱动的 inode
51   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
52   *                  一般在 open 的时候将 private_data 指向设备结构体。
53   * @return        : 0 成功;其他 失败
54   */
55  static int led_open(struct inode *inode, struct file *filp)
56  {
57      filp->private_data = &gpioled; /* 设置私有数据 */
58
59      /* 获取信号量,进入休眠状态的进程可以被信号打断 */
60      if (down_interruptible(&gpioled.sem)) {
61          return -ERESTARTSYS;
62      }
63  #if 0
64      down(&gpioled.sem);     /* 不能被信号打断 */
65  #endif
66
67      return 0;
68  }
......
114 /*
115  * @description   : 关闭/释放设备
116  * @param – filp  : 要关闭的设备文件(文件描述符)
117  * @return        : 0 成功;其他 失败
118  */
119 static int led_release(struct inode *inode, struct file *filp)
120 {
121     struct gpioled_dev *dev = filp->private_data;
122
123     up(&dev->sem);      /* 释放信号量，信号量值加 1 */
```

```
124
125     return 0;
126 }
127
128 /* 设备操作函数 */
129 static struct file_operations gpioled_fops = {
130     .owner = THIS_MODULE,
131     .open = led_open,
132     .read = led_read,
133     .write = led_write,
134     .release =  led_release,
135 };
136
137 /*
138  * @description   : 驱动入口函数
139  * @param         : 无
140  * @return        : 无
141  */
142 static int __init led_init(void)
143 {
144     int ret = 0;
145
146     /* 初始化信号量 */
147     sema_init(&gpioled.sem, 1);
......
204     return 0;
205 }
206
207 /*
208  * @description   : 驱动出口函数
209  * @param         : 无
210  * @return        : 无
211  */
212 static void __exit led_exit(void)
213 {
214     /* 注销字符设备驱动 */
215     cdev_del(&gpioled.cdev);/*  删除 cdev */
216     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
217
218     device_destroy(gpioled.class, gpioled.devid);
219     class_destroy(gpioled.class);
220 }
221
```

```
222 module_init(led_init);
223 module_exit(led_exit);
224 MODULE_LICENSE("GPL");
225 MODULE_AUTHOR("zuozhongkai");
```

第 14 行，要使用信号量必须添加<linux/semaphore.h>头文件。

第 43 行，在设备结构体中添加一个信号量成员变量 sem。

第 60~65 行，在 open 函数中申请信号量，可以使用 down 函数，也可以使用 down_interruptible 函数。如果信号量值大于 1 就表示可用，那么应用程序就会开始使用 LED 灯。如果信号量值为 0 就表示应用程序不能使用 LED 灯，此时应用程序就会进入到休眠状态。等到信号量值大于 1 的时候应用程序就会唤醒，申请信号量，获取 LED 灯使用权。

第 123 行，在 release 函数中调用 up 函数释放信号量，这样其他因为没有得到信号量而进入休眠状态的应用程序就会唤醒，获取信号量。

第 147 行，在驱动入口函数中调用 sema_init 函数初始化信号量 sem 的值为 1，相当于 sem 是个二值信号量。

总结一下，当信号量 sem 为 1 的时候表示 LED 灯还没有被使用，如果应用程序 A 要使用 LED 灯，先调用 open 函数打开/dev/gpioled，这个时候会获取信号量 sem，获取成功以后 sem 的值减 1 变为 0。如果此时应用程序 B 也要使用 LED 灯，调用 open 函数打开/dev/gpioled 就会因为信号量无效(值为 0)而进入休眠状态。当应用程序 A 运行完毕，调用 close 函数关闭/dev/gpioled 的时候就会释放信号量 sem，此时信号量 sem 的值就会加 1，变为 1。信号量 sem 再次有效，表示其他应用程序可以使用 LED 灯了，此时在休眠状态的应用程序 A 就会获取到信号量 sem，获取成功以后就开始使用 LED 灯。

3、编写测试 APP

测试 APP 使用 48.1.1 小节中的 atomicApp.c 即可，将 7_atomic 中的 atomicApp.c 文件到本例程中，并将 atomicApp.c 重命名为 semaApp.c 即可。

48.3.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 semaphore.o，Makefile 内容如下所示：

示例代码 48.3.2.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m  := semaphore.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 semaphore.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“semaphore.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 semaApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc semaApp.c -o semaApp
```

编译成功以后就会生成 semaApp 这个应用程序。

3、运行测试

将上一小节编译出来的 semaphore.ko 和 semaApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 semaphore.ko 驱动模块：

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe semaphore.ko //加载驱动
```

驱动加载成功以后就可以使用 semaApp 软件测试驱动是否工作正常，测试方法和 48.1.2 小节中一样，先输入如下命令让 semaApp 软件模拟占用 25S 的 LED 灯：

```
./ semaApp /dev/gpioled 1&         //打开 LED 灯
```

紧接着再输入如下命令关闭 LED 灯：

```
./ semaApp /dev/gpioled 0&         //关闭 LED 灯
```

注意两个命令都是运行在后台，第一条命令先获取到信号量，因此可以操作 LED 灯，将 LED 灯打开，并且占有 25S。第二条命令因为获取信号量失败而进入休眠状态，等待第一条命令运行完毕并释放信号量以后才拥有 LED 灯使用权，将 LED 灯关闭，运行结果如图 48.3.2.1 所示：

```
/lib/modules/4.1.15 # ./semaApp /dev/gpioled 1&
/lib/modules/4.1.15 # ./semaApp /dev/gpioled 0&
/lib/modules/4.1.15 # App running times:1
App running times:2
App running times:3
App running times:4
App running times:5
App running finished!
App running times:1
App running times:2
App running times:3
App running times:4
App running times:5
App running finished!
```


图 48.3.2.1 命令运行过程

如果要卸载驱动的话输入如下命令即可：

```
rmmod semaphore.ko
```

48.4 互斥体实验

前面我们使用原子操作、自旋锁和信号量实现了对 LED 灯的互斥访问，但是最适合互斥的就是互斥体 mutex 了。本节我们来学习一下如何使用 mutex 实现对 LED 灯的互斥访问。

48.4.1 实验程序编写

1、修改设备树文件

本章实验是在上一节实验的基础上完成的，同样不需要对设备树做任何的修改。

2、LED 驱动修改

本节实验在第上一节实验驱动文件 semaphore.c 的基础上修改而来。新建名为“10_mutex”的文件夹，然后在 10_mutex 文件夹里面创建 vscode 工程，工作区命名为“mutex”。将 9_semaphore 实验中的 semaphore.c 复制到 10_mutex 文件夹中，并且重命名为 mutex.c。将原来使用到信号量的地方换为 mutex 即可，其他的内容基本不变，完成以后的 mutex.c 文件内容如下所示(有省略)：

示例代码 48.4.1.1 mutex.c 文件代码

```
1   #include <linux/types.h>
......
17  #include <asm/io.h>
18  /***************************************************************
19  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20  文件名      : mutex.c
21  作者        : 左忠凯
22  版本        : V1.0
23  描述        : 互斥体实验，使用互斥体来实现对实现设备的互斥访问
24  其他        : 无
25  论坛        : www.openedv.com
26  日志        : 初版 V1.0 2019/7/18 左忠凯创建
27  ***************************************************************/
28  #define GPIOLED_CNT         1           /* 设备号个数   */
29  #define GPIOLED_NAME        "gpioled"   /* 名字         */
30  #define LEDOFF              0           /* 关灯         */
31  #define LEDON               1           /* 开灯         */
32  
33  /* gpioled 设备结构体 */
34  struct gpioled_dev{
35      dev_t devid;                /* 设备号           */
36      struct cdev cdev;           /* cdev             */
37      struct class *class;        /* 类               */
38      struct device *device;      /* 设备             */
39      int major;                  /* 主设备号         */
40      int minor;                  /* 次设备号         */
41      struct device_node  *nd;    /* 设备节点         */
42      int led_gpio;               /* led 所使用的 GPIO 编号*/
43      struct mutex lock;          /* 互斥体           */
44  };
45  
46  struct gpioled_dev gpioled; /* led 设备 */
47  
48  /*
49   * @description     : 打开设备
50   * @param – inode : 传递给驱动的 inode
51   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
52   *                  一般在 open 的时候将 private_data 指向设备结构体。
```

```
53   * @return         : 0 成功;其他 失败
54   */
55  static int led_open(struct inode *inode, struct file *filp)
56  {
57      filp->private_data = &gpioled; /* 设置私有数据 */
58  
59      /* 获取互斥体,可以被信号打断 */
60      if (mutex_lock_interruptible(&gpioled.lock)) {
61          return -ERESTARTSYS;
62      }
63  #if 0
64      mutex_lock(&gpioled.lock);  /* 不能被信号打断 */
65  #endif
66  
67      return 0;
68  }
......
114 /*
115  * @description   : 关闭/释放设备
116  * @param - filp  : 要关闭的设备文件(文件描述符)
117  * @return        : 0 成功;其他 失败
118  */
119 static int led_release(struct inode *inode, struct file *filp)
120 {
121     struct gpioled_dev *dev = filp->private_data;
122 
123     /* 释放互斥锁 */
124     mutex_unlock(&dev->lock);
125 
126     return 0;
127 }
128 
129 /* 设备操作函数 */
130 static struct file_operations gpioled_fops = {
131     .owner = THIS_MODULE,
132     .open = led_open,
133     .read = led_read,
134     .write = led_write,
135     .release =  led_release,
136 };
137 
138 /*
139  * @description   : 驱动入口函数
```

```
140  * @param          : 无
141  * @return         : 无
142  */
143 static int __init led_init(void)
144 {
145     int ret = 0;
146 
147     /* 初始化互斥体 */
148     mutex_init(&gpioled.lock);
......
205     return 0;
206 }
......
223 module_init(led_init);
224 module_exit(led_exit);
225 MODULE_LICENSE("GPL");
226 MODULE_AUTHOR("zuozhongkai");
```

第 43 行，定义互斥体 lock。

第 60~65 行，在 open 函数中调用 mutex_lock_interruptible 或者 mutex_lock 获取 mutex，成功的话就表示可以使用 LED 灯，失败的话就会进入休眠状态，和信号量一样。

第 124 行，在 release 函数中调用 mutex_unlock 函数释放 mutex，这样其他应用程序就可以获取 mutex 了。

第 148 行，在驱动入口函数中调用 mutex_init 初始化 mutex。

互斥体和二值信号量类似，只不过互斥体是专门用于互斥访问的。

3、编写测试 APP

测试 APP 使用 48.1.1 小节中的 atomicApp.c 即可，将 7_atomic 中的 atomicApp.c 文件到本例程中，并将 atomicApp.c 重命名为 mutexApp.c 即可。

48.4.2 运行测试

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 mutex.o，Makefile 内容如下所示：

示例代码 48.4.2.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m  := mutex.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 mutex.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“mutex.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 mutexApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc mutexApp.c -o mutexApp
```

编译成功以后就会生成 mutexApp 这个应用程序。

3、运行测试

将上一小节编译出来的 mutex.ko 和 mutexApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 mutex.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe mutex.ko       //加载驱动
```

驱动加载成功以后就可以使用 mutexApp 软件测试驱动是否工作正常，测试方法和 48.3.2 中测试信号量的方法一样。

如果要卸载驱动的话输入如下命令即可：

```
rmmod mutex.ko
```

第四十九章 Linux 按键输入实验

在前几章我们都是使用的 GPIO 输出功能，还没有用过 GPIO 输入功能，本章我们就来学习一下如果在 Linux 下编写 GPIO 输入驱动程序。I.MX6U-ALPHA 开发板上有一个按键，我们就使用此按键来完成 GPIO 输入驱动程序，同时利用第四十七章讲的原子操作来对按键值进行保护。

49.1 Linux 下按键驱动原理

按键驱动和 LED 驱动原理上来讲基本都是一样的，都是操作 GPIO，只不过一个是读取 GPIO 的高低电平，一个是从 GPIO 输出高低电平。本章实现我们实现按键输入，在驱动程序中使用一个整形变量来表示按键值，应用程序通过 read 函数来读取按键值，判断按键有没有按下。在这里，这个保存按键值的变量就是个共享资源，驱动程序要向其写入按键值，应用程序要读取按键值。所以我们要对其进行保护，对于整形变量而言我们首选的就是原子操作，使用原子操作对变量进行赋值以及读取。Linux 下的按键驱动原理很简单，接下来开始编写驱动。

注意，本章例程只是为了演示 Linux 下 GPIO 输入驱动的编写，实际中的按键驱动并不会采用本章中所讲解的方法，Linux 下的 input 子系统专门用于输入设备！

49.2 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

49.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 11_key。

49.3.1 修改设备树文件

1、添加 pinctrl 节点

I.MX6U-ALPHA 开发板上的 KEY 使用了 UART1_CTS_B 这个 PIN，打开 imx6ull-alientek-emmc.dts，在 iomuxc 节点的 imx6ul-evk 子节点下创建一个名为“pinctrl_key”的子节点，节点内容如下所示：

示例代码 49.3.1.1 按键 pinctrl 节点

```
1 pinctrl_key: keygrp {
2     fsl,pins = <
3         MX6UL_PAD_UART1_CTS_B__GPIO1_IO18      0xF080  /* KEY0 */
4     >;
5 };
```

第 3 行，将 GPIO_IO18 这个 PIN 复用为 GPIO1_IO18。

2、添加 KEY 设备节点

在根节点“/”下创建 KEY 节点，节点名为“key”，节点内容如下：

示例代码 49.3.1.2 创建 KEY 节点

```
1 key {
2     #address-cells = <1>;
3     #size-cells = <1>;
4     compatible = "atkalpha-key";
5     pinctrl-names = "default";
6     pinctrl-0 = <&pinctrl_key>;
7     key-gpio = <&gpio1 18 GPIO_ACTIVE_LOW>;     /* KEY0 */
8     status = "okay";
9 };
```

第 6 行，pinctrl-0 属性设置 KEY 所使用的 PIN 对应的 pinctrl 节点。

第 7 行，key-gpio 属性指定了 KEY 所使用的 GPIO。

3、检查 PIN 是否被其他外设使用

在本章实验中蜂鸣器使用的 PIN 为 UART1_CTS_B，因此先检查 PIN 为 UART1_CTS_B 这个 PIN 有没有被其他的 pinctrl 节点使用，如果有使用的话就要屏蔽掉，然后再检查 GPIO1_IO18 这个 GPIO 有没有被其他外设使用，如果有的话也要屏蔽掉。

设备树编写完成以后使用"make dtbs"命令重新编译设备树，然后使用新编译出来的 imx6ull-alientek-emmc.dtb 文件启动 Linux 系统。启动成功以后进入"/proc/device-tree"目录中查看"key"节点是否存在，如果存在的话就说明设备树基本修改成功(具体还要驱动验证)，结果如图 49.3.1.1 所示：

```
/sys/firmware/devicetree/base # ls
#address-cells                  interrupt-controller@00a01000
#size-cells                     key
aliases                         memory
alphaled                        model
backlight                       name
beep                            pxp_v4l2
chosen                          regulators
clocks                          reserved-memory
compatible                      soc
cpus                            sound
gpioled                         spi4
```

key子节点

图 46.3.1.1 key 子节点

49.3.2 按键驱动程序编写

设备树准备好以后就可以编写驱动程序了，新建名为"11_key"的文件夹，然后在 11_key 文件夹里面创建 vscode 工程，工作区命名为"key"。工程创建好以后新建 key.c 文件，在 key.c 里面输入如下内容：

示例代码 49.3.2.1 key.c 文件代码

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <linux/semaphore.h>
15 #include <asm/mach/map.h>
16 #include <asm/uaccess.h>
```

```
17 #include <asm/io.h>
18 /***************************************************************
19 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
20 文件名  : key.c
21 作者    : 左忠凯
22 版本    : V1.0
23 描述    : Linux 按键输入驱动实验
24 其他    : 无
25 论坛    : www.openedv.com
26 日志    : 初版 V1.0 2019/7/18 左忠凯创建
27 ***************************************************************/
28 #define KEY_CNT         1          /* 设备号个数   */
29 #define KEY_NAME        "key"      /* 名字         */
30 
31 /* 定义按键值 */
32 #define KEY0VALUE       0XF0       /* 按键值       */
33 #define INVAKEY         0X00       /* 无效的按键值 */
34 
35 /* key 设备结构体 */
36 struct key_dev{
37     dev_t devid;               /* 设备号       */
38     struct cdev cdev;          /* cdev         */
39     struct class *class;       /* 类           */
40     struct device *device;     /* 设备         */
41     int major;                 /* 主设备号     */
42     int minor;                 /* 次设备号     */
43     struct device_node  *nd;   /* 设备节点     */
44     int key_gpio;              /* key 所使用的 GPIO 编号       */
45     atomic_t keyvalue;         /* 按键值       */
46 };
47 
48 struct key_dev keydev;         /* key 设备 */
49 
50 /*
51  * @description    : 初始化按键 IO，open 函数打开驱动的时候
52  *                   初始化按键所使用的 GPIO 引脚。
53  * @param          : 无
54  * @return         : 无
55  */
56 static int keyio_init(void)
57 {
58     keydev.nd = of_find_node_by_path("/key");
59     if (keydev.nd== NULL) {
```

```
60          return -EINVAL;
61      }
62  
63      keydev.key_gpio = of_get_named_gpio(keydev.nd ,"key-gpio", 0);
64      if (keydev.key_gpio < 0) {
65          printk("can't get key0\r\n");
66          return -EINVAL;
67      }
68      printk("key_gpio=%d\r\n", keydev.key_gpio);
69  
70      /* 初始化 key 所使用的 IO */
71      gpio_request(keydev.key_gpio, "key0");     /* 请求 IO      */
72      gpio_direction_input(keydev.key_gpio);     /* 设置为输入    */
73      return 0;
74  }
75  
76  /*
77   * @description   : 打开设备
78   * @param – inode : 传递给驱动的 inode
79   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
80   *                  一般在 open 的时候将 private_data 指向设备结构体。
81   * @return        : 0 成功;其他 失败
82   */
83  static int key_open(struct inode *inode, struct file *filp)
84  {
85      int ret = 0;
86      filp->private_data = &keydev;   /* 设置私有数据     */
87  
88      ret = keyio_init();             /* 初始化按键 IO    */
89      if (ret < 0) {
90          return ret;
91      }
92  
93      return 0;
94  }
95  
96  /*
97   * @description   : 从设备读取数据
98   * @param - filp  : 要打开的设备文件(文件描述符)
99   * @param - buf   : 返回给用户空间的数据缓冲区
100  * @param - cnt   : 要读取的数据长度
101  * @param - offt  : 相对于文件首地址的偏移
102  * @return        : 读取的字节数，如果为负值，表示读取失败
```

```
103  */
104 static ssize_t key_read(struct file *filp, char __user *buf,
                             size_t cnt, loff_t *offt)
105 {
106     int ret = 0;
107     unsigned char value;
108     struct key_dev *dev = filp->private_data;
109
110     if (gpio_get_value(dev->key_gpio) == 0) { /* key0 按下    */
111         while(!gpio_get_value(dev->key_gpio)); /* 等待按键释放 */
112         atomic_set(&dev->keyvalue, KEY0VALUE);
113     } else {                                    /* 无效的按键值 */
114         atomic_set(&dev->keyvalue, INVAKEY);
115     }
116
117     value = atomic_read(&dev->keyvalue);        /* 保存按键值   */
118     ret = copy_to_user(buf, &value, sizeof(value));
119     return ret;
120 }
121
122
123 /* 设备操作函数 */
124 static struct file_operations key_fops = {
125     .owner = THIS_MODULE,
126     .open = key_open,
127     .read = key_read,
128 };
129
130 /*
131  * @description   : 驱动入口函数
132  * @param         : 无
133  * @return        : 无
134  */
135 static int __init mykey_init(void)
136 {
137     /* 初始化原子变量 */
138     atomic_set(&keydev.keyvalue, INVAKEY);
139
140     /* 注册字符设备驱动 */
141     /* 1、创建设备号    */
142     if (keydev.major) {          /*  定义了设备号    */
143         keydev.devid = MKDEV(keydev.major, 0);
144         register_chrdev_region(keydev.devid, KEY_CNT, KEY_NAME);
```

```
145     } else {                        /* 没有定义设备号  */
146         alloc_chrdev_region(&keydev.devid, 0, KEY_CNT, KEY_NAME);
147         keydev.major = MAJOR(keydev.devid); /* 获取分配号的主设备号 */
148         keydev.minor = MINOR(keydev.devid); /* 获取分配号的次设备号 */
149     }
150 
151     /* 2、初始化 cdev */
152     keydev.cdev.owner = THIS_MODULE;
153     cdev_init(&keydev.cdev, &key_fops);
154 
155     /* 3、添加一个 cdev */
156     cdev_add(&keydev.cdev, keydev.devid, KEY_CNT);
157 
158     /* 4、创建类 */
159     keydev.class = class_create(THIS_MODULE, KEY_NAME);
160     if (IS_ERR(keydev.class)) {
161         return PTR_ERR(keydev.class);
162     }
163 
164     /* 5、创建设备 */
165     keydev.device = device_create(keydev.class, NULL, keydev.devid,
                                     NULL, KEY_NAME);
166     if (IS_ERR(keydev.device)) {
167         return PTR_ERR(keydev.device);
168     }
169 
170     return 0;
171 }
172 
173 /*
174  * @description   : 驱动出口函数
175  * @param         : 无
176  * @return        : 无
177  */
178 static void __exit mykey_exit(void)
179 {
180     /* 注销字符设备驱动 */
181     cdev_del(&keydev.cdev);                         /*  删除 cdev */
182     unregister_chrdev_region(keydev.devid, KEY_CNT); /* 注销设备号 */
183 
184     device_destroy(keydev.class, keydev.devid);
185     class_destroy(keydev.class);
186 }
```

```
187
188 module_init(mykey_init);
189 module_exit(mykey_exit);
190 MODULE_LICENSE("GPL");
191 MODULE_AUTHOR("zuozhongkai");
```

第 36~46 行，结构体 key_dev 为按键的设备结构体，第 45 行的原子变量 keyvalue 用于记录按键值。

第 56~74 行，函数 keyio_init 用于初始化按键，从设备树中获取按键的 gpio 信息，然后设置为输入。将按键的初始化代码提取出来，将其作为独立的一个函数有利于提高程序的模块化设计。

第 83~94 行，key_open 函数通过调用 keyio_init 函数来始化按键所使用的 IO，应用程序每次打开按键驱动文件的时候都会初始化一次按键 IO。

第 104~120 行，key_read 函数，应用程序通过 read 函数读取按键值的时候此函数就会执行。第 110 行读取按键 IO 的电平，如果为 0 的话就表示按键按下了，如果按键按下的话第 111 行就等待按键释放。按键释放以后标记按键值为

第 135~171 行，驱动入口函数，第 138 行调用 atomic_set 函数初始化原子变量默认为无效值。

第 178~186 行，驱动出口函数。

key.c 文件代码很简单，重点就是 key_read 函数读取按键值，要对 keyvalue 进行保护。

49.3.3 编写测试 APP

新建名为 keyApp.c 的文件，然后输入如下所示内容：

示例代码 49.3.2.2 keyApp.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名    : keyApp.c
11 作者      : 左忠凯
12 版本      : V1.0
13 描述      : 按键输入测试应用程序
14 其他      : 无
15 使用方法 : ./keyApp /dev/key
16 论坛      : www.openedv.com
17 日志      : 初版 V1.0 2019/1/30 左忠凯创建
18 ***************************************************************/
19
20 /* 定义按键值 */
```

```
21 #define KEY0VALUE     0XF0
22 #define INVAKEY       0X00
23 
24 /*
25  * @description   : main 主程序
26  * @param - argc  : argv 数组元素个数
27  * @param - argv  : 具体参数
28  * @return        : 0 成功;其他 失败
29  */
30 int main(int argc, char *argv[])
31 {
32     int fd, ret;
33     char *filename;
34     unsigned char keyvalue;
35 
36     if(argc != 2){
37         printf("Error Usage!\r\n");
38         return -1;
39     }
40 
41     filename = argv[1];
42 
43     /* 打开 key 驱动 */
44     fd = open(filename, O_RDWR);
45     if(fd < 0){
46         printf("file %s open failed!\r\n", argv[1]);
47         return -1;
48     }
49 
50     /* 循环读取按键值数据！ */
51     while(1) {
52         read(fd, &keyvalue, sizeof(keyvalue));
53         if (keyvalue == KEY0VALUE) {    /* KEY0 */
54             printf("KEY0 Press, value = %#X\r\n", keyvalue);/* 按下 */
55         }
56     }
57 
58     ret= close(fd); /* 关闭文件 */
59     if(ret < 0){
60         printf("file %s close failed!\r\n", argv[1]);
61         return -1;
62     }
63     return 0;
```

```
64 }
```

第 51~56 行，循环读取/dev/key 文件，也就是循环读取按键值，并且将按键值打印出来。

49.4 运行测试

49.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 key.o，Makefile 内容如下所示：

示例代码 49.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
                rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := key.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 key.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“key.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 keyApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc keyApp.c -o keyApp
```

编译成功以后就会生成 keyApp 这个应用程序。

49.4.2 运行测试

将上一小节编译出来的 key.ko 和 keyApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 key.ko 驱动模块：

```
depmod              //第一次加载驱动的时候需要运行此命令
modprobe key.ko     //加载驱动
```

驱动加载成功以后如下命令来测试：

```
./keyApp /dev/key
```

输入上述命令以后终端显示如图 49.4.2.1 所示：

```
/lib/modules/4.1.15 # ./keyApp /dev/key
key_gpio=18
```

图 49.4.2.1 测试 APP 运行界面

按下开发板上的 KEY0 按键，keyApp 就会获取并且输出按键信息，如图 49.4.2.2 所示：

```
/lib/modules/4.1.15 # ./keyApp /dev/key
key_gpio=18
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
```

图 49.4.2.2 按键运行结果

从图 49.4.2.2 可以看出，当我们按下 KEY0 以后就会打印出“KEY0 Press, value = 0XF0”，表示按键按下。但是大家可能会发现，有时候按下一次 KEY0 但是会输出好几行“KEY0 Press, value = 0XF0”，这是因为我们的代码没有做按键消抖处理。

如果要卸载驱动的话输入如下命令即可：

```
rmmod key.ko
```

第五十章 Linux 内核定时器实验

定时器是我们最常用到的功能，一般用来完成定时功能，本章我们就来学习一下 Linux 内核提供的定时器 API 函数，通过这些定时器 API 函数我们可以完成很多要求定时的应用。Linux 内核也提供了短延时函数，比如微秒、纳秒、毫秒延时函数，本章我们就来学习一下这些和时间有关的功能。

50.1 Linux 时间管理和内核定时器简介

50.1.1 内核时间管理简介

学习过 UCOS 或 FreeRTOS 的同学应该知道，UCOS 或 FreeRTOS 是需要一个硬件定时器提供系统时钟，一般使用 Systick 作为系统时钟源。同理，Linux 要运行，也是需要一个系统时钟的，至于这个系统时钟是由哪个定时器提供的，笔者没有去研究过 Linux 内核，但是在 Cortex-A7 内核中有个通用定时器，在《Cortex-A7 Technical ReferenceManua.pdf》的"9:Generic Timer"章节有简单的讲解，关于这个通用定时器的详细内容，可以参考《ARM ArchitectureReference Manual ARMv7-A and ARMv7-R edition.pdf》的"chapter B8 The Generic Timer"章节。这个通用定时器是可选的，按照笔者学习 FreeRTOS 和 STM32 的经验，猜测 Linux 会将这个通用定时器作为 Linux 系统时钟源(前提是 SOC 得选配这个通用定时器)。具体是怎么做的笔者没有深入研究过，这里仅仅是猜测！不过对于我们 Linux 驱动编写者来说，不需要深入研究这些具体的实现，只需要掌握相应的 API 函数即可，除非你是内核编写者或者内核爱好者。

Linux 内核中有大量的函数需要时间管理，比如周期性的调度程序、延时程序、对于我们驱动编写者来说最常用的定时器。硬件定时器提供时钟源，时钟源的频率可以设置， 设置好以后就周期性的产生定时中断，系统使用定时中断来计时。中断周期性产生的频率就是系统频率，也叫做节拍率(tick rate)(有的资料也叫系统频率)，比如 1000Hz，100Hz 等等说的就是系统节拍率。系统节拍率是可以设置的，单位是 Hz，我们在编译 Linux 内核的时候可以通过图形化界面设置系统节拍率，按照如下路径打开配置界面：

```
-> Kernel Features
    -> Timer frequency (<choice> [=y])
```

选中"Timer frequency"，打开以后如图 50.1.1.1 所示：

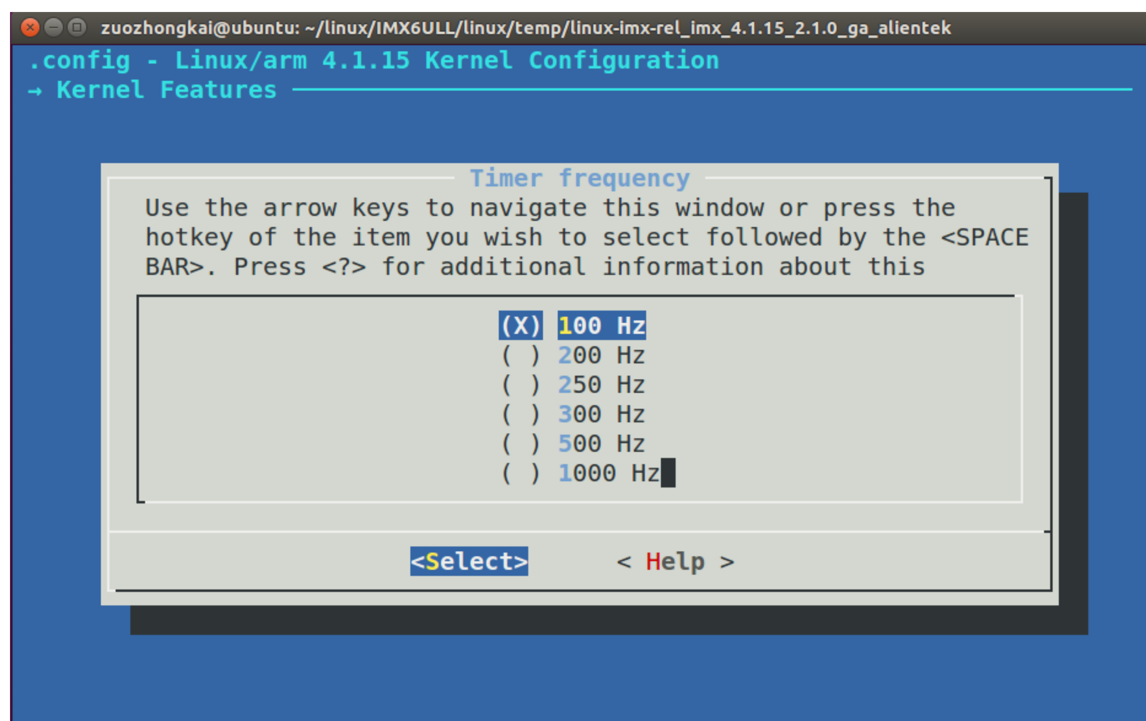


图 50.1.1.1 系统节拍率设置

从图 50.1.1.1 可以看出，可选的系统节拍率为 100Hz、200Hz、250Hz、300Hz、500Hz 和

1000Hz，默认情况下选择 100Hz。设置好以后打开 Linux 内核源码根目录下的.config 文件，在此文件中有如图 50.1.1.2 所示定义：

```
zuozhongkai@ubuntu: ~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek
508 CONFIG_PREEMPT_COUNT=y
509 CONFIG_HZ_FIXED=0
510 CONFIG_HZ_100=y
511 # CONFIG_HZ_200 is not set
512 # CONFIG_HZ_250 is not set
513 # CONFIG_HZ_300 is not set
514 # CONFIG_HZ_500 is not set
515 # CONFIG_HZ_1000 is not set
516 CONFIG_HZ=100
517 CONFIG_SCHED_HRTICK=y
518 CONFIG_AEABI=y
519 # CONFIG_OABI_COMPAT is not set
                                508,13        14%
```

系统节拍率

图 50.1.1.2 系统节拍率

图 50.1.1.2 中的 CONFIG_HZ 为 100，Linux 内核会使用 CONFIG_HZ 来设置自己的系统时钟。打开文件 include/asm-generic/param.h，有如下内容：

示例代码 50.1.1.1 include/asm-generic/param.h 文件代码段

```
6 # undef HZ
7 # define HZ                CONFIG_HZ
8 # define USER_HZ           100
9 # define CLOCKS_PER_SEC   (USER_HZ)
```

第 7 行定义了一个宏 HZ，宏 HZ 就是 CONFIG_HZ，因此 HZ=100，我们后面编写 Linux 驱动的时候会常常用到 HZ，因为 HZ 表示一秒的节拍数，也就是频率。

大多数初学者看到系统节拍率默认为 100Hz 的时候都会有疑问，怎么这么小？100Hz 是可选的节拍率里面最小的。为什么不选择大一点的呢？这里就引出了一个问题：高节拍率和低节拍率的优缺点：

①、高节拍率会提高系统时间精度，如果采用 100Hz 的节拍率，时间精度就是 10ms，采用 1000Hz 的话时间精度就是 1ms，精度提高了 10 倍。高精度时钟的好处有很多，对于那些对时间要求严格的函数来说，能够以更高的精度运行，时间测量也更加准确。

②、高节拍率会带导致中断的产生更加频繁，频繁的中断会加剧系统的负担，1000Hz 的 100Hz 的系统节拍率相比，系统要花费 10 倍的“精力”去处理中断。中断服务函数占用处理器的时间增加，但是现在的处理器性能都很强大，所以采用 1000Hz 的系统节拍率并不会增加太大的负载压力。根据自己的实际情况，选择合适的系统节拍率，本教程我们全部采用默认的 100Hz 系统节拍率。

Linux 内核使用全局变量 jiffies 来记录系统从启动以来的系统节拍数，系统启动的时候会将 jiffies 初始化为 0，jiffies 定义在文件 include/linux/jiffies.h 中，定义如下：

示例代码 50.1.1.2 include/jiffies.h 文件代码段

```
76 extern u64 __jiffy_data jiffies_64;
77 extern unsigned long volatile __jiffy_data jiffies;
```

第 76 行，定义了一个 64 位的 jiffies_64。

第 77 行，定义了一个 unsigned long 类型的 32 位的 jiffies。

jiffies_64 和 jiffies 其实是同一个东西，jiffies_64 用于 64 位系统，而 jiffies 用于 32 位系统。为了兼容不同的硬件，jiffies 其实就是 jiffies_64 的低 32 位，jiffies_64 和 jiffies 的结构如图 50.1.1.3 所示：

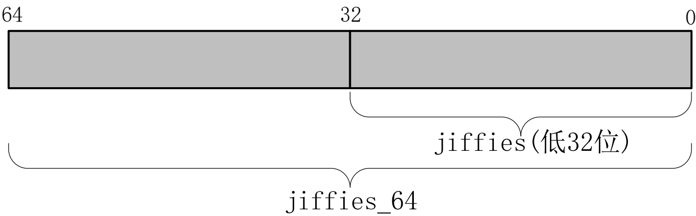


图 50.1.1.3 jiffies_64 和 jiffies 结构图

当我们访问 jiffies 的时候其实访问的是 jiffies_64 的低 32 位，使用 get_jiffies_64 这个函数可以获取 jiffies_64 的值。在 32 位的系统上读取 jiffies 的值，在 64 位的系统上 jiffes 和 jiffies_64 表示同一个变量，因此也可以直接读取 jiffies 的值。所以不管是 32 位的系统还是 64 位系统，都可以使用 jiffies。

前面说了 HZ 表示每秒的节拍数，jiffies 表示系统运行的 jiffies 节拍数，所以 jiffies/HZ 就是系统运行时间，单位为秒。不管是 32 位还是 64 位的 jiffies，都有溢出的风险，溢出以后会重新从 0 开始计数，相当于绕回来了，因此有些资料也将这个现象也叫做绕回。假如 HZ 为最大值 1000 的时候，32 位的 jiffies 只需要 49.7 天就发生了绕回，对于 64 为的 jiffies 来说大概需要 5.8 亿年才能绕回，因此 jiffies_64 的绕回忽略不计。处理 32 位 jiffies 的绕回显得尤为重要，Linux 内核提供了如表 50.1.1.1 所示的几个 API 函数来处理绕回。

函数	描述
time_after(unkown, known)	unkown 通常为 jiffies，known 通常是需要对比的值。
time_before(unkown, known)	
time_after_eq(unkown, known)	
time_before_eq(unkown, known)	

表 50.1.1.1 处理绕回的 API 函数

如果 unkown 超过 known 的话，time_after 函数返回真，否则返回假。如果 unkown 没有超过 known 的话 time_before 函数返回真，否则返回假。time_after_eq 函数和 time_after 函数类似，只是多了判断等于这个条件。同理，time_before_eq 函数和 time_before 函数也类似。比如我们要判断某段代码执行时间有没有超时，此时就可以使用如下所示代码：

示例代码 50.1.1.3 使用 jiffies 判断超时

```
1  unsigned long timeout;
2  timeout = jiffies + (2 * HZ);    /* 超时的时间点 */
3  
4  /*************************************
5    具体的代码
6   ************************************/
7  
8  /* 判断有没有超时 */
9  if(time_before(jiffies, timeout)) {
10  /* 超时发生 */
11 } else {
12  /* 超时未发生 */
13 }
```

timeout 就是超时时间点，比如我们要判断代码执行时间是不是超过了 2 秒，那么超时时间点就是 jiffies+(2*HZ)，如果 jiffies 大于 timeout 那就表示超时了，否则就是没有超时。第 4~6 行

就是具体的代码段。第 9 行通过函数 time_before 来判断 jiffies 是否小于 timeout，如果小于的话就表示没有超时。

为了方便开发，Linux 内核提供了几个 jiffies 和 ms、us、ns 之间的转换函数，如表 50.1.1.2 所示：

函数	描述
int jiffies_to_msecs(const unsigned long j)	将 jiffies 类型的参数 j 分别转换为对应的毫秒、微秒、纳秒。
int jiffies_to_usecs(const unsigned long j)	
u64 jiffies_to_nsecs(const unsigned long j)	
long msecs_to_jiffies(const unsigned int m)	将毫秒、微秒、纳秒转换为 jiffies 类型。
long usecs_to_jiffies(const unsigned int u)	
unsigned long nsecs_to_jiffies(u64 n)	

表 50.1.1.2 jiffies 和 ms、us、ns 之间的转换函数

50.1.2 内核定时器简介

定时器是一个很常用的功能，需要周期性处理的工作都要用到定时器。Linux 内核定时器采用系统时钟来实现，并不是我们在裸机篇中讲解的 PIT 等硬件定时器。Linux 内核定时器使用很简单，只需要提供超时时间(相当于定时值)和定时处理函数即可，当超时时间到了以后设置的定时处理函数就会执行，和我们使用硬件定时器的套路一样，只是使用内核定时器不需要做一大堆的寄存器初始化工作。在使用内核定时器的时候要注意一点，内核定时器并不是周期性运行的，超时以后就会自动关闭，因此如果想要实现周期性定时，那么就需要在定时处理函数中重新开启定时器。Linux 内核使用 timer_list 结构体表示内核定时器，timer_list 定义在文件 include/linux/timer.h 中，定义如下(省略掉条件编译)：

示例代码 50.1.2.1 timer_list 结构体

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;              /* 定时器超时时间，单位是节拍数   */
    struct tvec_base *base;

    void (*function)(unsigned long);    /* 定时处理函数                   */
    unsigned long data;                 /* 要传递给 function 函数的参数  */

    int slack;
};
```

要使用内核定时器首先要先定义一个 timer_list 变量，表示定时器，tiemr_list 结构体的 expires 成员变量表示超时时间，单位为节拍数。比如我们现在需要定义一个周期为 2 秒的定时器，那么这个定时器的超时时间就是 jiffies+(2*HZ)，因此 expires=jiffies+(2*HZ)。function 就是定时器超时以后的定时处理函数，我们要做的工作就放到这个函数里面，需要我们编写这个定时处理函数。

定义好定时器以后还需要通过一系列的 API 函数来初始化此定时器，这些函数如下：

1、init_timer 函数

init_timer 函数负责初始化 timer_list 类型变量，当我们定义了一个 timer_list 变量以后一定要先用 init_timer 初始化一下。init_timer 函数原型如下：

```
void init_timer(struct timer_list *timer)
```

函数参数和返回值含义如下：

timer：要初始化定时器。

返回值：没有返回值。

2、add_timer 函数

add_timer 函数用于向 Linux 内核注册定时器，使用 add_timer 函数向内核注册定时器以后，定时器就会开始运行，函数原型如下：

```
void add_timer(struct timer_list *timer)
```

函数参数和返回值含义如下：

timer：要注册的定时器。

返回值：没有返回值。

3、del_timer 函数

del_timer 函数用于删除一个定时器，不管定时器有没有被激活，都可以使用此函数删除。在多处理器系统上，定时器可能会在其他的处理器上运行，因此在调用 del_timer 函数删除定时器之前要先等待其他处理器的定时处理器函数退出。del_timer 函数原型如下：

```
int del_timer(struct timer_list * timer)
```

函数参数和返回值含义如下：

timer：要删除的定时器。

返回值：0，定时器还没被激活；1，定时器已经激活。

4、del_timer_sync 函数

del_timer_sync 函数是 del_timer 函数的同步版，会等待其他处理器使用完定时器再删除，del_timer_sync 不能使用在中断上下文中。del_timer_sync 函数原型如下所示：

```
int del_timer_sync(struct timer_list *timer)
```

函数参数和返回值含义如下：

timer：要删除的定时器。

返回值：0，定时器还没被激活；1，定时器已经激活。

5、mod_timer 函数

mod_timer 函数用于修改定时值，如果定时器还没有激活的话，mod_timer 函数会激活定时器！函数原型如下：

```
int mod_timer(struct timer_list *timer, unsigned long expires)
```

函数参数和返回值含义如下：

timer：要修改超时时间(定时值)的定时器。

expires：修改后的超时时间。

返回值：0，调用 mod_timer 函数前定时器未被激活；1，调用 mod_timer 函数前定时器已被激活。

关于内核定时器常用的 API 函数就讲这些，内核定时器一般的使用流程如下所示：

示例代码 50.1.2.2 内核定时器使用方法演示

```
1  struct timer_list timer; /* 定义定时器  */
2
3  /* 定时器回调函数 */
4  void function(unsigned long arg)
```

```
5  {
6      /*
7       * 定时器处理代码
8       */
9  
10     /* 如果需要定时器周期性运行的话就使用 mod_timer
11      * 函数重新设置超时值并且启动定时器。
12      */
13     mod_timer(&dev->timertest, jiffies + msecs_to_jiffies(2));
14 }
15 
16 /* 初始化函数 */
17 void init(void)
18 {
19     init_timer(&timer);                 /* 初始化定时器          */
20 
21     timer.function = function;          /* 设置定时处理函数      */
22     timer.expires=jffies + msecs_to_jiffies(2);/* 超时时间 2 秒 */
23     timer.data = (unsigned long)&dev;   /* 将设备结构体作为参数  */
24 
25  add_timer(&timer);                     /* 启动定时器            */
26 }
27 
28 /* 退出函数 */
29 void exit(void)
30 {
31     del_timer(&timer);  /* 删除定时器 */
32     /* 或者使用 */
33     del_timer_sync(&timer);
34 }
```

50.1.3 Linux 内核短延时函数

有时候我们需要在内核中实现短延时，尤其是在 Linux 驱动中。Linux 内核提供了毫秒、微秒和纳秒延时函数，这三个函数如表 50.1.3.1 所示：

函数	描述
void ndelay(unsigned long nsecs)	纳秒、微秒和毫秒延时函数。
void udelay(unsigned long usecs)	
void mdelay(unsigned long mseces)	

表 50.1.3.1 内核短延时函数

50.2 硬件原理图分析

本章使用通过设置一个定时器来实现周期性的闪烁 LED 灯，因此本章例程就使用到了一个

LED 灯，关于 LED 灯的硬件原理图参考参考 8.3 小节即可。

50.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 12_timer。

本章实验我们使用内核定时器周期性的点亮和熄灭开发板上的 LED 灯，LED 灯的闪烁周期由内核定时器来设置，测试应用程序可以控制内核定时器周期。

50.3.1 修改设备树文件

本章实验使用到了 LED 灯，LED 灯的设备树节点信息使用 45.4.1 小节创建的即可。

50.3.2 定时器驱动程序编写

新建名为“12_timer”的文件夹，然后在 12_timer 文件夹里面创建 vscode 工程，工作区命名为“timer”。工程创建好以后新建 timer.c 文件，在 timer.c 里面输入如下内容：

示例代码 50.3.2.1 timer.c 文件代码段

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <linux/semaphore.h>
15 #include <linux/timer.h>
16 #include <asm/mach/map.h>
17 #include <asm/uaccess.h>
18 #include <asm/io.h>
19 /***************************************************************
20 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
21 文件名       : timer.c
22 作者         : 左忠凯
23 版本         : V1.0
24 描述         : Linux 内核定时器实验
25 其他         : 无
26 论坛         : www.openedv.com
27 日志         : 初版 V1.0 2019/7/24 左忠凯创建
28 ***************************************************************/
```

```
29 #define TIMER_CNT          1          /* 设备号个数      */
30 #define TIMER_NAME         "timer"    /* 名字            */
31 #define CLOSE_CMD          (_IO(0XEF, 0x1))   /* 关闭定时器 */
32 #define OPEN_CMD           (_IO(0XEF, 0x2))   /* 打开定时器 */
33 #define SETPERIOD_CMD      (_IO(0XEF, 0x3))   /* 设置定时器周期命令 */
34 #define LEDON              1          /* 开灯                */
35 #define LEDOFF             0          /* 关灯                */
36 
37 /* timer 设备结构体 */
38 struct timer_dev{
39     dev_t devid;              /* 设备号              */
40     struct cdev cdev;         /* cdev                */
41     struct class *class;      /* 类                  */
42     struct device *device;    /* 设备                */
43     int major;                /* 主设备号            */
44     int minor;                /* 次设备号            */
45     struct device_node  *nd; /* 设备节点            */
46     int led_gpio;             /* key 所使用的 GPIO 编号    */
47     int timeperiod;           /* 定时周期,单位为 ms        */
48     struct timer_list timer; /* 定义一个定时器      */
49     spinlock_t lock;          /* 定义自旋锁          */
50 };
51 
52 struct timer_dev timerdev;  /* timer 设备          */
53 
54 /*
55  * @description   : 初始化 LED 灯 IO，open 函数打开驱动的时候
56  *                  初始化 LED 灯所使用的 GPIO 引脚。
57  * @param         : 无
58  * @return        : 无
59  */
60 static int led_init(void)
61 {
62     int ret = 0;
63 
64     timerdev.nd = of_find_node_by_path("/gpioled");
65     if (timerdev.nd== NULL) {
66         return -EINVAL;
67     }
68 
69     timerdev.led_gpio = of_get_named_gpio(timerdev.nd ,"led-gpio",
                                               0);
70     if (timerdev.led_gpio < 0) {
```

```
71          printk("can't get led\r\n");
72          return -EINVAL;
73      }
74  
75      /* 初始化 led 所使用的 IO */
76      gpio_request(timerdev.led_gpio, "led");     /* 请求 IO     */
77      ret = gpio_direction_output(timerdev.led_gpio, 1);
78      if(ret < 0) {
79          printk("can't set gpio!\r\n");
80      }
81      return 0;
82  }
83  
84  /*
85   * @description   : 打开设备
86   * @param – inode : 传递给驱动的 inode
87   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
88   *                 一般在 open 的时候将 private_data 指向设备结构体。
89   * @return        : 0 成功;其他 失败
90   */
91  static int timer_open(struct inode *inode, struct file *filp)
92  {
93      int ret = 0;
94      filp->private_data = &timerdev;     /* 设置私有数据     */
95  
96      timerdev.timeperiod = 1000;         /* 默认周期为 1s    */
97      ret = led_init();                   /* 初始化 LED IO    */
98      if (ret < 0) {
99          return ret;
100     }
101     return 0;
102 }
103 
104 /*
105  * @description   : ioctl 函数，
106  * @param – filp  : 要打开的设备文件(文件描述符)
107  * @param - cmd   : 应用程序发送过来的命令
108  * @param - arg   : 参数
109  * @return        : 0 成功;其他 失败
110  */
111 static long timer_unlocked_ioctl(struct file *filp, 
                                     unsigned int cmd, unsigned long arg)
112 {
```

```
113     struct timer_dev *dev =  (struct timer_dev *)filp->private_data;
114     int timerperiod;
115     unsigned long flags;
116 
117     switch (cmd) {
118         case CLOSE_CMD:           /* 关闭定时器      */
119             del_timer_sync(&dev->timer);
120             break;
121         case OPEN_CMD:            /* 打开定时器      */
122             spin_lock_irqsave(&dev->lock, flags);
123             timerperiod = dev->timeperiod;
124             spin_unlock_irqrestore(&dev->lock, flags);
125             mod_timer(&dev->timer, jiffies +
                          msecs_to_jiffies(timerperiod));
126             break;
127         case SETPERIOD_CMD:       /* 设置定时器周期   */
128             spin_lock_irqsave(&dev->lock, flags);
129             dev->timeperiod = arg;
130             spin_unlock_irqrestore(&dev->lock, flags);
131             mod_timer(&dev->timer, jiffies + msecs_to_jiffies(arg));
132             break;
133         default:
134             break;
135     }
136     return 0;
137 }
138 
139 /* 设备操作函数 */
140 static struct file_operations timer_fops = {
141     .owner = THIS_MODULE,
142     .open = timer_open,
143     .unlocked_ioctl = timer_unlocked_ioctl,
144 };
145 
146 /* 定时器回调函数 */
147 void timer_function(unsigned long arg)
148 {
149     struct timer_dev *dev = (struct timer_dev *)arg;
150     static int sta = 1;
151     int timerperiod;
152     unsigned long flags;
153 
154     sta = !sta;         /* 每次都取反，实现 LED 灯反转 */
```

```
155     gpio_set_value(dev->led_gpio, sta);
156 
157     /* 重启定时器 */
158     spin_lock_irqsave(&dev->lock, flags);
159     timerperiod = dev->timeperiod;
160     spin_unlock_irqrestore(&dev->lock, flags);
161     mod_timer(&dev->timer, jiffies +
                 msecs_to_jiffies(dev->timeperiod));
162  }
163 
164 /*
165  * @description   : 驱动入口函数
166  * @param         : 无
167  * @return        : 无
168  */
169 static int __init timer_init(void)
170 {
171     /* 初始化自旋锁 */
172     spin_lock_init(&timerdev.lock);
173 
174     /* 注册字符设备驱动 */
175     /* 1、创建设备号 */
176     if (timerdev.major) {       /*  定义了设备号 */
177         timerdev.devid = MKDEV(timerdev.major, 0);
178         register_chrdev_region(timerdev.devid, TIMER_CNT,
                                        TIMER_NAME);
179     } else {                      /* 没有定义设备号 */
180         alloc_chrdev_region(&timerdev.devid, 0, TIMER_CNT,
                                        TIMER_NAME);
181         timerdev.major = MAJOR(timerdev.devid);   /* 获取主设备号 */
182         timerdev.minor = MINOR(timerdev.devid);   /* 获取次设备号 */
183     }
184 
185     /* 2、初始化 cdev */
186     timerdev.cdev.owner = THIS_MODULE;
187     cdev_init(&timerdev.cdev, &timer_fops);
188 
189     /* 3、添加一个 cdev */
190     cdev_add(&timerdev.cdev, timerdev.devid, TIMER_CNT);
191 
192     /* 4、创建类 */
193     timerdev.class = class_create(THIS_MODULE, TIMER_NAME);
194     if (IS_ERR(timerdev.class)) {
```

```
195         return PTR_ERR(timerdev.class);
196     }
197 
198     /* 5、创建设备 */
199     timerdev.device = device_create(timerdev.class, NULL,
                                      timerdev.devid, NULL, TIMER_NAME);
200     if (IS_ERR(timerdev.device)) {
201         return PTR_ERR(timerdev.device);
202     }
203 
204     /* 6、初始化 timer，设置定时器处理函数,还未设置周期，所有不会激活定时器 */
205     init_timer(&timerdev.timer);
206     timerdev.timer.function = timer_function;
207     timerdev.timer.data = (unsigned long)&timerdev;
208     return 0;
209 }
210 
211 /*
212  * @description  : 驱动出口函数
213  * @param        : 无
214  * @return       : 无
215  */
216 static void __exit timer_exit(void)
217 {
218 
219     gpio_set_value(timerdev.led_gpio, 1); /* 卸载驱动的时候关闭 LED */
220     del_timer_sync(&timerdev.timer);      /* 删除 timer   */
221 #if 0
222     del_timer(&timerdev.tiemr);
223 #endif
224 
225     /* 注销字符设备驱动 */
226     cdev_del(&timerdev.cdev);             /*  删除 cdev */
227     unregister_chrdev_region(timerdev.devid, TIMER_CNT);
228 
229     device_destroy(timerdev.class, timerdev.devid);
230     class_destroy(timerdev.class);
231 }
232 
233 module_init(timer_init);
234 module_exit(timer_exit);
235 MODULE_LICENSE("GPL");
236 MODULE_AUTHOR("zuozhongkai");
```

第 38~50 行，定时器设备结构体，在 48 行定义了一个定时器成员变量 timer。

第 60~82 行，LED 灯初始化函数，从设备树中获取 LED 灯信息，然后初始化相应的 IO。

第 91~102 行，函数 timer_open，对应应用程序的 open 函数，应用程序调用 open 函数打开/dev/timer 驱动文件的时候此函数就会执行。此函数设置文件私有数据为 timerdev，并且初始化定时周期默认为 1 秒，最后调用 led_init 函数初始化 LED 所使用的 IO。

第 111~137 行，函数 timer_unlocked_ioctl，对应应用程序的 ioctl 函数，应用程序调用 ioctl 函数向驱动发送控制信息，此函数响应并执行。此函数有三个参数：filp，cmd 和 arg，其中 filp 是对应的设备文件，cmd 是应用程序发送过来的命令信息，arg 是应用程序发送过来的参数，在本章例程中 arg 参数表示定时周期。

一共有三种命令 CLOSE_CMD，OPEN_CMD 和 SETPERIOD_CMD，这三个命令分别为关闭定时器、打开定时器、设置定时周期。这三个命令的左右如下：

CLOSE_CMD：关闭定时器命令，调用 del_timer_sync 函数关闭定时器。

OPEN_CMD：打开定时器命令，调用 mod_timer 函数打开定时器，定时周期为 timerdev 的 timeperiod 成员变量，定时周期默认是 1 秒。

SETPERIOD_CMD：设置定时器周期命令，参数 arg 就是新的定时周期，设置 timerdev 的 timeperiod 成员变量为 arg 所表示定时周期指。并且使用 mod_timer 重新打开定时器，使定时器以新的周期运行。

第 140~144 行，定时器驱动操作函数集 timer_fops。

第 147~162 行，函数 timer_function，定时器服务函数，此函有一个参数 arg，在本例程中 arg 参数就是 timerdev 的地址，这样通过 arg 参数就可以访问到设备结构体。当定时周期到了以后此函数就会被调用。在此函数中将 LED 灯的状态取反，实现 LED 灯闪烁的效果。因为内核定时器不是循环的定时器，执行一次以后就结束了，因此在 161 行又调用了 mod_timer 函数重新开启定时器。

第 169~209 行，函数 timer_init，驱动入口函数。在第 205~207 行初始化定时器，设置定时器的定时处理函数为 timer_function，另外设置要传递给 timer_function 函数的参数为 timerdev。在此函数中并没有调用 timer_add 函数来开启定时器，因此定时器默认是关闭的，除非应用程序发送打开命令。

第 216~231 行，驱动出口函数，在 219 行关闭 LED，也就是卸载驱动以后 LED 处于熄灭状态。第 220 行调用 del_timer_sync 函数删除定时器，也可以使用 del_timer 函数。

50.3.3 编写测试 APP

测试 APP 我们要实现的内容如下：

①、运行 APP 以后提示我们输入要测试的命令，输入 1 表示关闭定时器、输入 2 表示打开定时器，输入 3 设置定时器周期。

②、如果要设置定时器周期的话，需要让用户输入要设置的周期值，单位为毫秒。

新建名为 timerApp.c 的文件，然后输入如下所示内容：

示例代码 50.3.2.2 timerApp.c 文件代码段
```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
```

```
7  #include "string.h"
8  #include "linux/ioctl.h"
9  /***************************************************************
10 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
11 文件名        : timerApp.c
12 作者          : 左忠凯
13 版本          : V1.0
14 描述          : 定时器测试应用程序
15 其他          : 无
16 使用方法      : ./timertest /dev/timer 打开测试 App
17 论坛          : www.openedv.com
18 日志          : 初版 V1.0 2019/7/24 左忠凯创建
19 ***************************************************************/
20 
21 /* 命令值 */
22 #define CLOSE_CMD         (_IO(0XEF, 0x1))    /* 关闭定时器        */
23 #define OPEN_CMD          (_IO(0XEF, 0x2))    /* 打开定时器        */
24 #define SETPERIOD_CMD     (_IO(0XEF, 0x3))    /* 设置定时器周期命令 */
25 
26 /*
27  * @description    : main 主程序
28  * @param - argc   : argv 数组元素个数
29  * @param - argv   : 具体参数
30  * @return         : 0 成功;其他 失败
31  */
32 int main(int argc, char *argv[])
33 {
34     int fd, ret;
35     char *filename;
36     unsigned int cmd;
37     unsigned int arg;
38     unsigned char str[100];
39 
40     if (argc != 2) {
41         printf("Error Usage!\r\n");
42         return -1;
43     }
44 
45     filename = argv[1];
46 
47     fd = open(filename, O_RDWR);
48     if (fd < 0) {
49         printf("Can't open file %s\r\n", filename);
```

```
50          return -1;
51      }
52
53      while (1) {
54          printf("Input CMD:");
55          ret = scanf("%d", &cmd);
56          if (ret != 1) {             /* 参数输入错误         */
57              gets(str);              /* 防止卡死             */
58          }
59
60          if(cmd == 1)                /* 关闭 LED 灯          */
61              cmd = CLOSE_CMD;
62          else if(cmd == 2)           /* 打开 LED 灯          */
63              cmd = OPEN_CMD;
64          else if(cmd == 3) {
65              cmd = SETPERIOD_CMD;    /* 设置周期值           */
66              printf("Input Timer Period:");
67              ret = scanf("%d", &arg);
68              if (ret != 1) {         /* 参数输入错误         */
69                  gets(str);          /* 防止卡死             */
70              }
71          }
72          ioctl(fd, cmd, arg);        /* 控制定时器的打开和关闭 */
73      }
74      close(fd);
75  }
```

第 22~24 行，命令值。

第 53~73 行，while(1)循环，让用户输入要测试的命令，然后通过第 72 行的 ioctl 函数发送给驱动程序。如果是设置定时器周期命令 SETPERIOD_CMD，那么 ioctl 函数的 arg 参数就是用户输入的周期值。

50.4 运行测试

50.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 timer.o，Makefile 内容如下所示：

示例代码 50.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := timer.o
```

```
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 timer.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“timer.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 timerApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc timerApp.c -o timerApp
```

编译成功以后就会生成 timerApp 这个应用程序。

50.4.2 运行测试

将上一小节编译出来的 timer.ko 和 timerApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 timer.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe timer.ko       //加载驱动
```

驱动加载成功以后如下命令来测试：

```
./timerApp   /dev/timer
```

输入上述命令以后终端提示输入命令，如图 50.4.2.1 所示：

```
/lib/modules/4.1.15 # ./timerApp /dev/timer
Input CMD:
```

图 50.4.2.1 输入命令

输入“2”，打开定时器，此时 LED 灯就会以默认的 1 秒周期开始闪烁。在输入“3”来设置定时周期，根据提示输入要设置的周期值，如图 50.4.2.2 所示：

```
Input CMD:3
Input Timer Period:
```

图 50.4.2.2 设置周期值

输入“500”，表示设置定时器周期值为 500ms，设置好以后 LED 灯就会以 500ms 为间隔，开始闪烁。最后可以通过输入“1”来关闭定时器，如果要卸载驱动的话输入如下命令即可：

```
rmmod timer.ko
```

第五十一章 Linux 中断实验

不管是裸机实验还是 Linux 下的驱动实验，中断都是频繁使用的功能，关于 I.MX6U 的中断原理已经在第十七章做了详细的讲解，在裸机中使用中断我们需要做一大堆的工作，比如配置寄存器，使能 IRQ 等等。Linux 内核提供了完善的中断框架，我们只需要申请中断，然后注册中断处理函数即可，使用非常方便，不需要一系列复杂的寄存器配置。本章我们就来学习一下如何在 Linux 下使用中断。

51.1 Linux 中断简介

51.1.1 Linux 中断 API 函数

先来回顾一下裸机实验里面中断的处理方法：

①、使能中断，初始化相应的寄存器。

②、注册中断服务函数，也就是向 irqTable 数组的指定标号处写入中断服务函数

②、中断发生以后进入 IRQ 中断服务函数，在 IRQ 中断服务函数在数组 irqTable 里面查找具体的中断处理函数，找到以后执行相应的中断处理函数。

在 Linux 内核中也提供了大量的中断相关的 API 函数，我们来看一下这些跟中断有关的 API 函数：

1、中断号

每个中断都有一个中断号，通过中断号即可区分不同的中断，有的资料也把中断号叫做中断线。在 Linux 内核中使用一个 int 变量表示中断号，关于中断号我们已经在第十七章讲解过了。

2、request_irq 函数

在 Linux 内核中要想使用某个中断是需要申请的，request_irq 函数用于申请中断，request_irq 函数可能会导致睡眠，因此不能在中断上下文或者其他禁止睡眠的代码段中使用 request_irq 函数。request_irq 函数会激活(使能)中断，所以不需要我们手动去使能中断，request_irq 函数原型如下：

```
int request_irq(unsigned int        irq,
                irq_handler_t       handler,
                unsigned long       flags,
                const char          *name,
                void                *dev)
```

函数参数和返回值含义如下：

irq：要申请中断的中断号。

handler：中断处理函数，当中断发生以后就会执行此中断处理函数。

flags：中断标志，可以在文件 include/linux/interrupt.h 里面查看所有的中断标志，这里我们介绍几个常用的中断标志，如表 51.1.1.1 所示：

标志	描述
IRQF_SHARED	多个设备共享一个中断线，共享的所有中断都必须指定此标志。如果使用共享中断的话，request_irq 函数的 dev 参数就是唯一区分他们的标志。
IRQF_ONESHOT	单次中断，中断执行一次就接触。
IRQF_TRIGGER_NONE	无触发。
IRQF_TRIGGER_RISING	上升沿触发。
IRQF_TRIGGER_FALLING	下降沿触发。
IRQF_TRIGGER_HIGH	高电平触发。
IRQF_TRIGGER_LOW	低电平触发。

表 51.1.1.1 常用的中断标志

比如 I.MX6U-ALPHA 开发板上的 KEY0 使用 GPIO1_IO03，按下 KEY0 以后为低电平，因此可以设置为下降沿触发，也就是将 flags 设置为 IRQF_TRIGGER_FALLING。表 51.1.1.1 中的这些标志可以通过“|”来实现多种组合。

name：中断名字，设置以后可以在/proc/interrupts 文件中看到对应的中断名字。

dev：如果将 flags 设置为 IRQF_SHARED 的话，dev 用来区分不同的中断，一般情况下将 dev 设置为设备结构体，dev 会传递给中断处理函数 irq_handler_t 的第二个参数。

返回值：0 中断申请成功，其他负值 中断申请失败，如果返回-EBUSY 的话表示中断已经被申请了。

3、free_irq 函数

使用中断的时候需要通过 request_irq 函数申请，使用完成以后就要通过 free_irq 函数释放掉相应的中断。如果中断不是共享的，那么 free_irq 会删除中断处理函数并且禁止中断。free_irq 函数原型如下所示：

```
void free_irq(unsigned int    irq,
              void            *dev)
```

函数参数和返回值含义如下：

irq：要释放的中断。

dev：如果中断设置为共享(IRQF_SHARED)的话，此参数用来区分具体的中断。共享中断只有在释放最后中断处理函数的时候才会被禁止掉。

返回值：无。

4、中断处理函数

使用 request_irq 函数申请中断的时候需要设置中断处理函数，中断处理函数格式如下所示：

```
irqreturn_t (*irq_handler_t) (int, void *)
```

第一个参数是要中断处理函数要相应的中断号。第二个参数是一个指向 void 的指针，也就是个通用指针，需要与 request_irq 函数的 dev 参数保持一致。用于区分共享中断的不同设备，dev 也可以指向设备数据结构。中断处理函数的返回值为 irqreturn_t 类型，irqreturn_t 类型定义如下所示：

示例代码 51.1.1.1 irqreturn_t 结构

```
10 enum irqreturn {
11     IRQ_NONE             = (0 << 0),
12     IRQ_HANDLED          = (1 << 0),
13     IRQ_WAKE_THREAD      = (1 << 1),
14 };
15
16 typedef enum irqreturn irqreturn_t;
```

可以看出 irqreturn_t 是个枚举类型，一共有三种返回值。一般中断服务函数返回值使用如下形式：

```
return IRQ_RETVAL(IRQ_HANDLED)
```

5、中断使能与禁止函数

常用的中断使用和禁止函数如下所示：

```
void enable_irq(unsigned int irq)
void disable_irq(unsigned int irq)
```

enable_irq 和 disable_irq 用于使能和禁止指定的中断，irq 就是要禁止的中断号。disable_irq

函数要等到当前正在执行的中断处理函数执行完才返回，因此使用者需要保证不会产生新的中断，并且确保所有已经开始执行的中断处理程序已经全部退出。在这种情况下，可以使用另外一个中断禁止函数：

```
void disable_irq_nosync(unsigned int irq)
```

disable_irq_nosync 函数调用以后立即返回，不会等待当前中断处理程序执行完毕。上面三个函数都是使能或者禁止某一个中断，有时候我们需要关闭当前处理器的整个中断系统，也就是在学习 STM32 的时候常说的关闭全局中断，这个时候可以使用如下两个函数：

```
local_irq_enable()
local_irq_disable()
```

local_irq_enable 用于使能当前处理器中断系统，local_irq_disable 用于禁止当前处理器中断系统。假如 A 任务调用 local_irq_disable 关闭全局中断 10S，当关闭了 2S 的时候 B 任务开始运行，B 任务也调用 local_irq_disable 关闭全局中断 3S，3 秒以后 B 任务调用 local_irq_enable 函数将全局中断打开了。此时才过去 2+3=5 秒的时间，然后全局中断就被打开了，此时 A 任务要关闭 10S 全局中断的愿望就破灭了，然后 A 任务就“生气了”，结果很严重，可能系统都要被 A 任务整崩溃。为了解决这个问题，B 任务不能直接简单粗暴的通过 local_irq_enable 函数来打开全局中断，而是将中断状态恢复到以前的状态，要考虑到别的任务的感受，此时就要用到下面两个函数：

```
local_irq_save(flags)
local_irq_restore(flags)
```

这两个函数是一对，local_irq_save 函数用于禁止中断，并且将中断状态保存在 flags 中。local_irq_restore 用于恢复中断，将中断到 flags 状态。

51.1.2 上半部与下半部

在有些资料中也将上半部和下半部称为顶半部和底半部，都是一个意思。我们在使用 request_irq 申请中断的时候注册的中断服务函数属于中断处理的上半部，只要中断触发，那么中断处理函数就会执行。我们都知道中断处理函数一定要快点执行完毕，越短越好，但是现实往往是残酷的，有些中断处理过程就是比较费时间，我们必须要对其进行处理，缩小中断处理函数的执行时间。比如电容触摸屏通过中断通知 SOC 有触摸事件发生，SOC 响应中断，然后通过 IIC 接口读取触摸坐标值并将其上报给系统。但是我们都知道 IIC 的速度最高也只有 400Kbit/S，所以在中断中通过 IIC 读取数据就会浪费时间。我们可以将通过 IIC 读取触摸数据的操作暂后执行，中断处理函数仅仅相应中断，然后清除中断标志位即可。这个时候中断处理过程就分为了两部分：

上半部：上半部就是中断处理函数，那些处理过程比较快，不会占用很长时间的处理就可以放在上半部完成。

下半部：如果中断处理过程比较耗时，那么就将这些比较耗时的代码提出来，交给下半部去执行，这样中断处理函数就会快进快出。

因此，Linux 内核将中断分为上半部和下半部的主要目的就是实现中断处理函数的快进快出，那些对时间敏感、执行速度块的操作可以放到中断处理函数中，也就是上半部。剩下的所有工作都可以放到下半部去执行，比如在上半部将数据拷贝到内存中，关于数据的具体处理就可以放到下半部去执行。至于哪些代码属于上半部，哪些代码属于下半部并没有明确的规定，一切根据实际使用情况去判断，这个就很考验驱动编写人员的功底了。这里有一些可以借鉴的参考点：

①、如果要处理的内容不希望被其他中断打断，那么可以放到上半部。

②、如果要处理的任务对时间敏感，可以放到上半部。

③、如果要处理的任务与硬件有关，可以放到上半部

④、除了上述三点以外的其他任务，优先考虑放到下半部。

上半部处理很简单，直接编写中断处理函数就行了，关键是下半部该怎么做呢？Linux 内核提供了多种下半部机制，接下来我们来学习一下这些下半部机制。

1、软中断

一开始 Linux 内核提供了"bootom half"机制来实现下半部，简称"BH"。后面引入了软中断和 tasklet 来替代"BH"机制，完全可以使用软中断和 taskled 来替代 BH，从 2.5 版本的 Linux 内核开始 BH 已经被抛弃了。Linux 内核使用结构体 softirq_action 表示软中断， softirq_action 结构体定义在文件 include/linux/interrupt.h 中，内容如下：

示例代码 51.1.2.1 softirq_action 结构体

```
433 struct softirq_action
434 {
435     void    (*action)(struct softirq_action *);
436 };
```

在 kernel/softirq.c 文件中一共定义了 10 个软中断，如下所示：

示例代码 51.1.2.2 softirq_vec 数组

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

NR_SOFTIRQS 是枚举类型，定义在文件 include/linux/interrupt.h 中，定义如下：

示例代码 51.1.2.3 softirq_vec 数组

```
enum
{
    HI_SOFTIRQ=0,           /* 高优先级软中断     */
    TIMER_SOFTIRQ,          /* 定时器软中断       */
    NET_TX_SOFTIRQ,         /* 网络数据发送软中断 */
    NET_RX_SOFTIRQ,         /* 网络数据接收软中断 */
    BLOCK_SOFTIRQ,
    BLOCK_IOPOLL_SOFTIRQ,
    TASKLET_SOFTIRQ,        /* tasklet 软中断     */
    SCHED_SOFTIRQ,          /* 调度软中断         */
    HRTIMER_SOFTIRQ,        /* 高精度定时器软中断 */
    RCU_SOFTIRQ,            /* RCU 软中断         */

    NR_SOFTIRQS
};
```

可以看出，一共有 10 个软中断，因此 NR_SOFTIRQS 为 10，因此数组 softirq_vec 有 10 个元素。softirq_action 结构体中的 action 成员变量就是软中断的服务函数，数组 softirq_vec 是个全局数组，因此所有的 CPU(对于 SMP 系统而言)都可以访问到，每个 CPU 都有自己的触发和控制机制，并且只执行自己所触发的软中断。但是各个 CPU 所执行的软中断服务函数确是相同的，都是数组 softirq_vec 中定义的 action 函数。要使用软中断，必须先使用 open_softirq 函数注册对应的软中断处理函数，open_softirq 函数原型如下：

```
void open_softirq(int nr,   void (*action)(struct softirq_action *))
```

函数参数和返回值含义如下：

nr：要开启的软中断，在示例代码 51.1.2.3 中选择一个。

action：软中断对应的处理函数。

返回值：没有返回值。

注册好软中断以后需要通过 raise_softirq 函数触发，raise_softirq 函数原型如下：

```
void raise_softirq(unsigned int nr)
```

函数参数和返回值含义如下：

nr：要触发的软中断，在示例代码 51.1.2.3 中选择一个。

返回值：没有返回值。

软中断必须在编译的时候静态注册！Linux 内核使用 softirq_init 函数初始化软中断，softirq_init 函数定义在 kernel/softirq.c 文件里面，函数内容如下：

示例代码 51.1.2.4 softirq_init 函数内容

```
634 void __init softirq_init(void)
635 {
636     int cpu;
637 
638     for_each_possible_cpu(cpu) {
639         per_cpu(tasklet_vec, cpu).tail =
640             &per_cpu(tasklet_vec, cpu).head;
641         per_cpu(tasklet_hi_vec, cpu).tail =
642             &per_cpu(tasklet_hi_vec, cpu).head;
643     }
644 
645     open_softirq(TASKLET_SOFTIRQ, tasklet_action);
646     open_softirq(HI_SOFTIRQ, tasklet_hi_action);
647 }
```

从示例代码 51.1.2.4 可以看出，softirq_init 函数默认会打开 TASKLET_SOFTIRQ 和 HI_SOFTIRQ。

2、tasklet

tasklet 是利用软中断来实现的另外一种下半部机制，在软中断和 tasklet 之间，建议大家使用 tasklet。Linux 内核使用结构体

示例代码 51.1.2.5 tasklet_struct 结构体

```
484 struct tasklet_struct
485 {
486     struct tasklet_struct *next;    /* 下一个 tasklet          */
487     unsigned long state;            /* tasklet 状态           */
488     atomic_t count;                 /* 计数器，记录对 tasklet 的引用数 */
489     void (*func)(unsigned long);    /* tasklet 执行的函数       */
490     unsigned long data;             /* 函数 func 的参数         */
491 };
```

第 488 行的 func 函数就是 tasklet 要执行的处理函数，用户定义函数内容，相当于中断处理函数。如果要使用 tasklet，必须先定义一个 tasklet，然后使用 tasklet_init 函数初始化 tasklet，taskled_init 函数原型如下：

```
void tasklet_init(struct tasklet_struct    *t,
```

```
        void (*func)(unsigned long),
        unsigned long    data);
```

函数参数和返回值含义如下：

t：要初始化的 tasklet

func：tasklet 的处理函数。

data：要传递给 func 函数的参数

返回值：没有返回值。

也可以使用宏 DECLARE_TASKLET 来一次性完成 tasklet 的定义和初始化，DECLARE_TASKLET 定义在 include/linux/interrupt.h 文件中，定义如下:

```
DECLARE_TASKLET(name, func, data)
```

其中 name 为要定义的 tasklet 名字，这个名字就是一个 tasklet_struct 类型的时候变量，func 就是 tasklet 的处理函数，data 是传递给 func 函数的参数。

在上半部，也就是中断处理函数中调用 tasklet_schedule 函数就能使 tasklet 在合适的时间运行，tasklet_schedule 函数原型如下：

```
void tasklet_schedule(struct tasklet_struct *t)
```

函数参数和返回值含义如下：

t：要调度的 tasklet，也就是 DECLARE_TASKLET 宏里面的 name。

返回值：没有返回值。

关于 tasklet 的参考使用示例如下所示：

示例代码 51.1.2.7 tasklet 使用示例

```
/* 定义 taselet             */
struct tasklet_struct testtasklet;

/* tasklet 处理函数         */
void testtasklet_func(unsigned long data)
{
    /* tasklet 具体处理内容 */
}

/* 中断处理函数 */
irqreturn_t test_handler(int irq, void *dev_id)
{
    ......
    /* 调度 tasklet         */
    tasklet_schedule(&testtasklet);
    ......
}

/* 驱动入口函数             */
static int __init xxxx_init(void)
{
    ......
    /* 初始化 tasklet       */
```

```
    tasklet_init(&testtasklet, testtasklet_func, data);
    /* 注册中断处理函数      */
    request_irq(xxx_irq, test_handler, 0, "xxx", &xxx_dev);
    ......
}
```

2、工作队列

工作队列是另外一种下半部执行方式，工作队列在进程上下文执行，工作队列将要推后的工作交给一个内核线程去执行，因为工作队列工作在进程上下文，因此工作队列允许睡眠或重新调度。因此如果你要推后的工作可以睡眠那么就可以选择工作队列，否则的话就只能选择软中断或 tasklet。

Linux 内核使用 work_struct 结构体表示一个工作，内容如下(省略掉条件编译)：

示例代码 51.1.2.8 work_struct 结构体

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;          /* 工作队列处理函数  */
};
```

这些工作组织成工作队列，工作队列使用 workqueue_struct 结构体表示，内容如下(省略掉条件编译)：

示例代码 51.1.2.9 workqueue_struct 结构体

```
struct workqueue_struct {
    struct list_head    pwqs;
    struct list_head    list;
    struct mutex        mutex;
    int         work_color;
    int         flush_color;
    atomic_t        nr_pwqs_to_flush;
    struct wq_flusher   *first_flusher;
    struct list_head    flusher_queue;
    struct list_head    flusher_overflow;
    struct list_head    maydays;
    struct worker       *rescuer;
    int         nr_drainers;
    int         saved_max_active;
    struct workqueue_attrs  *unbound_attrs;
    struct pool_workqueue   *dfl_pwq;
    char            name[WQ_NAME_LEN];
    struct rcu_head     rcu;
    unsigned int        flags ____cacheline_aligned;
    struct pool_workqueue __percpu *cpu_pwqs;
    struct pool_workqueue __rcu *numa_pwq_tbl[];
};
```

Linux 内核使用工作者线程(worker thred)来处理工作队列中的各个工作，Linux 内核使用 worker 结构体表示工作者线程，worker 结构体内容如下：

示例代码 51.1.2.10 worker 结构体

```
struct worker {
    union {
        struct list_head    entry;
        struct hlist_node   hentry;
    };
    struct work_struct  *current_work;
    work_func_t     current_func;
    struct pool_workqueue   *current_pwq;
    bool            desc_valid;
    struct list_head    scheduled;
    struct task_struct  *task;
    struct worker_pool  *pool;
    struct list_head    node;
    unsigned long       last_active;
    unsigned int        flags;
    int         id;
    char            desc[WORKER_DESC_LEN];
    struct workqueue_struct *rescue_wq;
};
```

从示例代码 51.1.2.10 可以看出，每个 worker 都有一个工作队列，工作者线程处理自己工作队列中的所有工作。在实际的驱动开发中，我们只需要定义工作(work_struct)即可，关于工作队列和工作者线程我们基本不用去管。简单创建工作很简单，直接定义一个 work_struct 结构体变量即可，然后使用 INIT_WORK 宏来初始化工作，INIT_WORK 宏定义如下：

```
#define INIT_WORK(_work, _func)
```

_work 表示要初始化的工作，_func 是工作对应的处理函数。

也可以使用 DECLARE_WORK 宏一次性完成工作的创建和初始化，宏定义如下：

```
#define DECLARE_WORK(n, f)
```

n 表示定义的工作(work_struct)，f 表示工作对应的处理函数。

和 tasklet 一样，工作也是需要调度才能运行的，工作的调度函数为 schedule_work，函数原型如下所示：

```
bool schedule_work(struct work_struct *work)
```

函数参数和返回值含义如下：

work： 要调度的工作。

返回值： 0 成功，其他值 失败。

关于工作队列的参考使用示例如下所示：

示例代码 51.1.2.11 工作队列使用示例

```
/* 定义工作(work)          */
struct  work_struct testwork;

/* work 处理函数           */
```

```
void testwork_func_t(struct work_struct *work);
{
    /* work 具体处理内容     */
}

/* 中断处理函数            */
irqreturn_t test_handler(int irq, void *dev_id)
{
    ......
    /* 调度 work            */
    schedule_work(&testwork);
    ......
}

/* 驱动入口函数            */
static int __init xxxx_init(void)
{
    ......
    /* 初始化 work          */
    INIT_WORK(&testwork, testwork_func_t);
    /* 注册中断处理函数      */
    request_irq(xxx_irq, test_handler, 0, "xxx", &xxx_dev);
    ......
}
```

51.1.3 设备树中断信息节点

如果使用设备树的话就需要在设备树中设置好中断属性信息，Linux 内核通过读取设备树中的中断属性信息来配置中断。对于中断控制器而言，设备树绑定信息参考文档 Documentation/devicetree/bindings/arm/gic.txt。打开 imx6ull.dtsi 文件，其中的 intc 节点就是 I.MX6ULL 的中断控制器节点，节点内容如下所示：

示例代码 51.1.3.1 中断控制器 intc 节点

```
1 intc: interrupt-controller@00a01000 {
2     compatible = "arm,cortex-a7-gic";
3     #interrupt-cells = <3>;
4     interrupt-controller;
5     reg = <0x00a01000 0x1000>,
6           <0x00a02000 0x100>;
7 };
```

第 2 行，compatible 属性值为“arm,cortex-a7-gic”在 Linux 内核源码中搜索“arm,cortex-a7-gic”即可找到 GIC 中断控制器驱动文件。

第 3 行，#interrupt-cells 和#address-cells、#size-cells 一样。表示此中断控制器下设备的 cells 大小，对于设备而言，会使用 interrupts 属性描述中断信息，#interrupt-cells 描述了 interrupts 属

性的 cells 大小，也就是一条信息有几个 cells。每个 cells 都是 32 位整形值，对于 ARM 处理的 GIC 来说，一共有 3 个 cells，这三个 cells 的含义如下：

第一个 cells：中断类型，0 表示 SPI 中断，1 表示 PPI 中断。

第二个 cells：中断号，对于 SPI 中断来说中断号的范围为 0~987，对于 PPI 中断来说中断号的范围为 0~15。

第三个 cells：标志，bit[3:0]表示中断触发类型，为 1 的时候表示上升沿触发，为 2 的时候表示下降沿触发，为 4 的时候表示高电平触发，为 8 的时候表示低电平触发。bit[15:8]为 PPI 中断的 CPU 掩码。

第 4 行，interrupt-controller 节点为空，表示当前节点是中断控制器。

对于 gpio 来说，gpio 节点也可以作为中断控制器，比如 imx6ull.dtsi 文件中的 gpio5 节点内容如下所示：

示例代码 51.1.3.2 gpio5 设备节点

```
1  gpio5: gpio@020ac000 {
2      compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
3      reg = <0x020ac000 0x4000>;
4      interrupts = <GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>,
5                   <GIC_SPI 75 IRQ_TYPE_LEVEL_HIGH>;
6      gpio-controller;
7      #gpio-cells = <2>;
8      interrupt-controller;
9      #interrupt-cells = <2>;
10 };
```

第 4 行，interrupts 描述中断源信息，对于 gpio5 来说一共有两条信息，中断类型都是 SPI，触发电平都是 IRQ_TYPE_LEVEL_HIGH。不同之处在于中断源，一个是 74，一个是 75，打开可以打开《IMX6ULL 参考手册》的“Chapter 3 Interrupts and DMA Events”章节，找到表 3-1，有如图 50.1.3.1 所示的内容：

IRQ	Interrupt Source	LOGIC	Interrupt Description
74	gpio5	-	Combined interrupt indication for GPIO5 signal 0 throughout 15
75	gpio5	-	Combined interrupt indication for GPIO5 signal 16 throughout 31

图 50.1.3.1 中断表

从图 50.1.3.1 可以看出，GPIO5 一共用了 2 个中断号，一个是 74，一个是 75。其中 74 对应 GPIO5_IO00~GPIO5_IO15 这低 16 个 IO，75 对应 GPIO5_IO16~GPIOI5_IO31 这高 16 位 IO。

第 8 行，interrupt-controller 表明了 gpio5 节点也是个中断控制器，用于控制 gpio5 所有 IO 的中断。

第 9 行，将#interrupt-cells 修改为 2。

打开 imx6ull-alientek-emmc.dts 文件，找到如下所示内容：

示例代码 51.1.3.3 fxls8471 设备节点

```
1 fxls8471@1e {
2     compatible = "fsl,fxls8471";
3     reg = <0x1e>;
4     position = <0>;
5     interrupt-parent = <&gpio5>;
```

```
6     interrupts = <0 8>;
7 };
```

fxls8471 是 NXP 官方的 6ULL 开发板上的一个磁力计芯片，fxls8471 有一个中断引脚链接到了 I.MX6ULL 的 SNVS_TAMPER0 因脚上，这个引脚可以复用为 GPIO5_IO00。

第 5 行，interrupt-parent 属性设置中断控制器，这里使用 gpio5 作为中断控制器。

第 6 行，interrupts 设置中断信息，0 表示 GPIO5_IO00，8 表示低电平触发。

简单总结一下与中断有关的设备树属性信息：

①、#interrupt-cells，指定中断源的信息 cells 个数。

②、interrupt-controller，表示当前节点为中断控制器。

③、interrupts，指定中断号，触发方式等。

④、interrupt-parent，指定父中断，也就是中断控制器。

51.1.4 获取中断号

编写驱动的时候需要用到中断号，我们用到中断号，中断信息已经写到了设备树里面，因此可以通过 irq_of_parse_and_map 函数从 interups 属性中提取到对应的设备号，函数原型如下：

```
unsigned int irq_of_parse_and_map(struct device_node    *dev,
                                  int                    index)
```

函数参数和返回值含义如下：

dev：设备节点。

index：索引号，interrupts 属性可能包含多条中断信息，通过 index 指定要获取的信息。

返回值：中断号。

如果使用 GPIO 的话，可以使用 gpio_to_irq 函数来获取 gpio 对应的中断号，函数原型如下：

```
int gpio_to_irq(unsigned int gpio)
```

函数参数和返回值含义如下：

gpio：要获取的 GPIO 编号。

返回值：GPIO 对应的中断号中断号。

51.2 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

51.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 13_irq。

本章实验我们驱动 I.MX6U-ALPHA 开发板上的 KEY0 按键，不过我们采用中断的方式，并且采用定时器来实现按键消抖，应用程序读取按键值并且通过终端打印出来。通过本章我们可以学习到 Linux 内核中断的使用方法，以及对 Linux 内核定时器的回顾。

51.3.1 修改设备树文件

本章实验使用到了按键 KEY0，按键 KEY0 使用中断模式，因此需要在“key”节点下添加中断相关属性，添加完成以后的“key”节点内容如下所示：

示例代码 51.3.1.1 key 节点信息

```
1 key {
```

```
2     #address-cells = <1>;
3     #size-cells = <1>;
4     compatible = "atkalpha-key";
5     pinctrl-names = "default";
6     pinctrl-0 = <&pinctrl_key>;
7     key-gpio = <&gpio1 18 GPIO_ACTIVE_LOW>; /* KEY0 */
8     interrupt-parent = <&gpio1>;
9     interrupts = <18 IRQ_TYPE_EDGE_BOTH>;   /* FALLING RISING */
10    status = "okay";
11 };
```

第 8 行，设置 interrupt-parent 属性值为“gpio1”，因为 KEY0 所使用的 GPIO 为 GPIO1_IO18，也就是设置 KEY0 的 GPIO 中断控制器为 gpio1。

第 9 行，设置 interrupts 属性，也就是设置中断源，第一个 cells 的 18 表示 GPIO1 组的 18 号 IO。IRQ_TYPE_EDGE_BOTH 定义在文件 include/linux/irq.h 中，定义如下：

示例代码 51.3.1.2 中断线状态

```
76 enum {
77     IRQ_TYPE_NONE              = 0x00000000,
78     IRQ_TYPE_EDGE_RISING       = 0x00000001,
79     IRQ_TYPE_EDGE_FALLING      = 0x00000002,
80     IRQ_TYPE_EDGE_BOTH         = (IRQ_TYPE_EDGE_FALLING |
                                     IRQ_TYPE_EDGE_RISING),
81     IRQ_TYPE_LEVEL_HIGH        = 0x00000004,
82     IRQ_TYPE_LEVEL_LOW         = 0x00000008,
83     IRQ_TYPE_LEVEL_MASK        = (IRQ_TYPE_LEVEL_LOW |
                                     IRQ_TYPE_LEVEL_HIGH),
......
100 };
```

从示例代码 51.3.1.2 中可以看出，IRQ_TYPE_EDGE_BOTH 表示上升沿和下降沿同时有效，相当于 KEY0 按下和释放都会触发中断。

设备树编写完成以后使用“make dtbs”命令重新编译设备树，然后使用新编译出来的 imx6ull-alientek-emmc.dtb 文件启动 Linux 系统。

51.3.2 按键中断驱动程序编写

新建名为“13_irq”的文件夹，然后在 13_irq 文件夹里面创建 vscode 工程，工作区命名为“imx6uirq”。工程创建好以后新建 imx6uirq.c 文件，在 imx6uirq.c 里面输入如下内容：

示例代码 51.3.2.1 imx6uirq.c 文件代码

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
```

```
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of.h>
12  #include <linux/of_address.h>
13  #include <linux/of_gpio.h>
14  #include <linux/semaphore.h>
15  #include <linux/timer.h>
16  #include <linux/of_irq.h>
17  #include <linux/irq.h>
18  #include <asm/mach/map.h>
19  #include <asm/uaccess.h>
20  #include <asm/io.h>
21  /***************************************************************
22  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
23  文件名    : imx6uirq.c
24  作者      : 左忠凯
25  版本      : V1.0
26  描述      : Linux 中断驱动实验
27  其他      : 无
28  论坛      : www.openedv.com
29  日志      : 初版 V1.0 2019/7/26 左忠凯创建
30  ***************************************************************/
31  #define IMX6UIRQ_CNT        1           /* 设备号个数       */
32  #define IMX6UIRQ_NAME       "imx6uirq" /* 名字             */
33  #define KEY0VALUE           0X01        /* KEY0 按键值      */
34  #define INVAKEY             0XFF        /* 无效的按键值     */
35  #define KEY_NUM             1           /* 按键数量         */
36  
37  /* 中断 IO 描述结构体 */
38  struct irq_keydesc {
39      int gpio;                           /* gpio             */
40      int irqnum;                         /* 中断号           */
41      unsigned char value;                /* 按键对应的键值   */
42      char name[10];                      /* 名字             */
43      irqreturn_t (*handler)(int, void *);   /* 中断服务函数 */
44  };
45  
46  /* imx6uirq 设备结构体 */
47  struct imx6uirq_dev{
48      dev_t devid;                /* 设备号           */
49      struct cdev cdev;           /* cdev             */
50      struct class *class;        /* 类               */
```

```
51      struct device *device;   /* 设备                    */
52      int major;               /* 主设备号                */
53      int minor;               /* 次设备号                */
54      struct device_node  *nd; /* 设备节点                */
55      atomic_t keyvalue;       /* 有效的按键键值          */
56      atomic_t releasekey;     /* 标记是否完成一次完成的按键*/
57      struct timer_list timer; /* 定义一个定时器*/
58      struct irq_keydesc irqkeydesc[KEY_NUM];   /* 按键描述数组 */
59      unsigned char curkeynum;                  /* 当前的按键号 */
60  };
61
62  struct imx6uirq_dev imx6uirq;   /* irq 设备 */
63
64  /* @description      : 中断服务函数，开启定时器，延时 10ms，
65   *                     定时器用于按键消抖。
66   * @param - irq       : 中断号
67   * @param - dev_id    : 设备结构。
68   * @return            : 中断执行结果
69   */
70  static irqreturn_t key0_handler(int irq, void *dev_id)
71  {
72      struct imx6uirq_dev *dev = (struct imx6uirq_dev *)dev_id;
73
74      dev->curkeynum = 0;
75      dev->timer.data = (volatile long)dev_id;
76      mod_timer(&dev->timer, jiffies + msecs_to_jiffies(10));
77      return IRQ_RETVAL(IRQ_HANDLED);
78  }
79
80  /* @description   : 定时器服务函数，用于按键消抖，定时器到了以后
81   *                  再次读取按键值，如果按键还是处于按下状态就表示按键有效。
82   * @param – arg   : 设备结构变量
83   * @return        : 无
84   */
85  void timer_function(unsigned long arg)
86  {
87      unsigned char value;
88      unsigned char num;
89      struct irq_keydesc *keydesc;
90      struct imx6uirq_dev *dev = (struct imx6uirq_dev *)arg;
91
92      num = dev->curkeynum;
93      keydesc = &dev->irqkeydesc[num];
```

```
94 
95      value = gpio_get_value(keydesc->gpio);     /* 读取 IO 值    */
96      if(value == 0){                            /* 按下按键      */
97          atomic_set(&dev->keyvalue, keydesc->value);
98      }
99      else{                                      /* 按键松开      */
100         atomic_set(&dev->keyvalue, 0x80 | keydesc->value);
101         atomic_set(&dev->releasekey, 1);       /* 标记松开按键 */
102     }
103 }
104 
105 /*
106  * @description   : 按键 IO 初始化
107  * @param         : 无
108  * @return        : 无
109  */
110 static int keyio_init(void)
111 {
112     unsigned char i = 0;
113     char name[10];
114     int ret = 0;
115 
116     imx6uirq.nd = of_find_node_by_path("/key");
117     if (imx6uirq.nd== NULL){
118         printk("key node not find!\r\n");
119         return -EINVAL;
120     }
121 
122     /* 提取 GPIO */
123     for (i = 0; i < KEY_NUM; i++) {
124         imx6uirq.irqkeydesc[i].gpio = of_get_named_gpio(imx6uirq.nd,
                                                    "key-gpio", i);
125         if (imx6uirq.irqkeydesc[i].gpio < 0) {
126             printk("can't get key%d\r\n", i);
127         }
128     }
129 
130     /* 初始化 key 所使用的 IO，并且设置成中断模式 */
131     for (i = 0; i < KEY_NUM; i++) {
132         memset(imx6uirq.irqkeydesc[i].name, 0, sizeof(name));
133         sprintf(imx6uirq.irqkeydesc[i].name, "KEY%d", i);
134         gpio_request(imx6uirq.irqkeydesc[i].gpio, name);
135         gpio_direction_input(imx6uirq.irqkeydesc[i].gpio);
```

```
136         imx6uirq.irqkeydesc[i].irqnum = irq_of_parse_and_map(
                                               imx6uirq.nd, i);
137 #if 0
138         imx6uirq.irqkeydesc[i].irqnum = gpio_to_irq(
                                               imx6uirq.irqkeydesc[i].gpio);
139 #endif
140         printk("key%d:gpio=%d, irqnum=%d\r\n",i,
                                             imx6uirq.irqkeydesc[i].gpio,
141                                          imx6uirq.irqkeydesc[i].irqnum);
142     }
143     /* 申请中断 */
144     imx6uirq.irqkeydesc[0].handler = key0_handler;
145     imx6uirq.irqkeydesc[0].value = KEY0VALUE;
146 
147     for (i = 0; i < KEY_NUM; i++) {
148         ret = request_irq(imx6uirq.irqkeydesc[i].irqnum,
                              imx6uirq.irqkeydesc[i].handler,
                              IRQF_TRIGGER_FALLING|IRQF_TRIGGER_RISING,
                              imx6uirq.irqkeydesc[i].name, &imx6uirq);
149         if(ret < 0){
150             printk("irq %d request failed!\r\n",
                        imx6uirq.irqkeydesc[i].irqnum);
151             return -EFAULT;
152         }
153     }
154 
155     /* 创建定时器 */
156     init_timer(&imx6uirq.timer);
157     imx6uirq.timer.function = timer_function;
158     return 0;
159 }
160 
161 /*
162  * @description   : 打开设备
163  * @param – inode : 传递给驱动的 inode
164  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
165  *                  一般在 open 的时候将 private_data 指向设备结构体。
166  * @return        : 0 成功;其他 失败
167  */
168 static int imx6uirq_open(struct inode *inode, struct file *filp)
169 {
170     filp->private_data = &imx6uirq; /* 设置私有数据 */
171     return 0;
```

```
172 }
173
174  /*
175   * @description  : 从设备读取数据
176   * @param - filp : 要打开的设备文件(文件描述符)
177   * @param - buf  : 返回给用户空间的数据缓冲区
178   * @param - cnt  : 要读取的数据长度
179   * @param - offt : 相对于文件首地址的偏移
180   * @return       : 读取的字节数，如果为负值，表示读取失败
181   */
182 static ssize_t imx6uirq_read(struct file *filp, char __user *buf,
                                size_t cnt, loff_t *offt)
183 {
184     int ret = 0;
185     unsigned char keyvalue = 0;
186     unsigned char releasekey = 0;
187     struct imx6uirq_dev *dev = (struct imx6uirq_dev *)
                                  filp->private_data;
188
189     keyvalue = atomic_read(&dev->keyvalue);
190     releasekey = atomic_read(&dev->releasekey);
191
192     if (releasekey) {                     /* 有按键按下   */
193         if (keyvalue & 0x80) {
194             keyvalue &= ~0x80;
195             ret = copy_to_user(buf, &keyvalue, sizeof(keyvalue));
196         } else {
197             goto data_error;
198         }
199         atomic_set(&dev->releasekey, 0);   /* 按下标志清零 */
200     } else {
201         goto data_error;
202     }
203     return 0;
204
205 data_error:
206     return -EINVAL;
207 }
208
209 /* 设备操作函数 */
210 static struct file_operations imx6uirq_fops = {
211     .owner = THIS_MODULE,
212     .open = imx6uirq_open,
```

```
213     .read = imx6uirq_read,
214 };
215 
216 /*
217  * @description  : 驱动入口函数
218  * @param        : 无
219  * @return       : 无
220  */
221 static int __init imx6uirq_init(void)
222 {
223     /* 1、构建设备号 */
224     if (imx6uirq.major) {
225         imx6uirq.devid = MKDEV(imx6uirq.major, 0);
226         register_chrdev_region(imx6uirq.devid, IMX6UIRQ_CNT,
                                   IMX6UIRQ_NAME);
227     } else {
228         alloc_chrdev_region(&imx6uirq.devid, 0, IMX6UIRQ_CNT,
                                   IMX6UIRQ_NAME);
229         imx6uirq.major = MAJOR(imx6uirq.devid);
230         imx6uirq.minor = MINOR(imx6uirq.devid);
231     }
232 
233     /* 2、注册字符设备  */
234     cdev_init(&imx6uirq.cdev, &imx6uirq_fops);
235     cdev_add(&imx6uirq.cdev, imx6uirq.devid, IMX6UIRQ_CNT);
236 
237     /* 3、创建类       */
238     imx6uirq.class = class_create(THIS_MODULE, IMX6UIRQ_NAME);
239     if (IS_ERR(imx6uirq.class)) {
240         return PTR_ERR(imx6uirq.class);
241     }
242 
243     /* 4、创建设备     */
244     imx6uirq.device = device_create(imx6uirq.class, NULL,
                                 imx6uirq.devid, NULL, IMX6UIRQ_NAME);
245     if (IS_ERR(imx6uirq.device)) {
246         return PTR_ERR(imx6uirq.device);
247     }
248 
249     /* 5、始化按键     */
250     atomic_set(&imx6uirq.keyvalue, INVAKEY);
251     atomic_set(&imx6uirq.releasekey, 0);
252     keyio_init();
```

```
253     return 0;
254 }
255 
256 /*
257  * @description     : 驱动出口函数
258  * @param           : 无
259  * @return          : 无
260  */
261 static void __exit imx6uirq_exit(void)
262 {
263     unsigned i = 0;
264     /* 删除定时器 */
265     del_timer_sync(&imx6uirq.timer);
266 
267     /* 释放中断 */
268     for (i = 0; i < KEY_NUM; i++) {
269         free_irq(imx6uirq.irqkeydesc[i].irqnum, &imx6uirq);
270     }
271     cdev_del(&imx6uirq.cdev);
272     unregister_chrdev_region(imx6uirq.devid, IMX6UIRQ_CNT);
273     device_destroy(imx6uirq.class, imx6uirq.devid);
274     class_destroy(imx6uirq.class);
275 }
276 
277 module_init(imx6uirq_init);
278 module_exit(imx6uirq_exit);
279 MODULE_LICENSE("GPL");
280 MODULE_AUTHOR("zuozhongkai");
```

第 38~43 行，结构体 irq_keydesc 为按键的中断描述结构体，gpio 为按键 GPIO 编号，irqnum 为按键 IO 对应的中断号，value 为按键对应的键值，name 为按键名字，handler 为按键中断服务函数。使用 irq_keydesc 结构体即可描述一个按键中断。

第 47~60 行，结构体 imx6uirq_dev 为本例程设备结构体，第 55 行的 keyvalue 保存按键值，第 56 行的 releasekey 表示按键是否被释放，如果按键被释放表示发生了一次完整的按键过程。第 57 行的 timer 为按键消抖定时器，使用定时器进行按键消抖的原理已经在 19.1 小节讲解过了。第 58 行的数组 irqkeydesc 为按键信息数组，数组元素个数就是开发板上的按键个数，I.MX6U-ALIPHA 开发板上只有一个按键，因此 irqkeydesc 数组只有一个元素。第 59 行的 curkeynum 表示当前按键。

第 62 行，定义设备结构体变量 imx6uirq。

第 70~78 行，key0_handler 函数，按键 KEY0 中断处理函数，参数 dev_id 为设备结构体，也就是 imx6uirq。第 74 行设置 curkeynum=0，表示当前按键为 KEY0，第 76 行使用 mod_timer 函数启动定时器，定时器周期为 10ms。

第 85~103，timer_function 函数，定时器定时处理函数，参数 arg 是设备结构体，也就是 imx6uirq，在此函数中读取按键值。第 95 行通过 gpio_get_value 函数读取按键值。如果为 0 的

话就表示按键被按下去了，按下去的话就设置 imx6uirq 结构体的 keyvalue 成员变量为按键的键值，比如 KEY0 按键的话按键值就是 KEY0VALUE=0。如果按键值为 1 的话表示按键被释放了，按键释放了的话就将 imx6uirq 结构体的 keyvalue 成员变量的最高位置 1，表示按键值有效，也就是将 keyvalue 与 0x80 进行或运算，表示按键松开了，并且设置 imx6uirq 结构体的 releasekey 成员变量为 1，表示按键释放，一次有效的按键过程发生。

第 110~159 行，keyio_init 函数，按键 IO 初始化函数，在驱动入口函数里面会调用 keyio_init 来初始化按键 IO。第 131~142 行轮流初始化所有的按键，包括申请 IO、设置 IO 为输入模式、从设备树中获取 IO 的中断号等等。第 136 行通过 irq_of_parse_and_map 函数从设备树中获取按键 IO 对应的中断号。也可以使用 gpio_to_irq 函数将某个 IO 设置为中断状态，并且返回其中断号。第 144 和 145 行设置 KEY0 按键对应的按键中断处理函数为 key0_handler、KEY0 的按键值为 KEY0VALUE。第 147~153 行轮流调用 request_irq 函数申请中断号，设置中断触发模式为 IRQF_TRIGGER_FALLING 和 IRQF_TRIGGER_RISING，也就是上升沿和下降沿都可以触发中断。最后，第 156 行初始化定时器，并且设置定时器的定时处理函数。

第 168~172 行，imx6uirq_open 函数，对应应用程序的 open 函数。

第 182~207 行，imx6uirq_read 函数，对应应用程序的 read 函数。此函数向应用程序返回按键值。首先判断 imx6uirq 结构体的 releasekey 成员变量值是否为 1，如果为 1 的话表示又一次有效按键发生，否则的话就直接返回-EINVAL。当有按键事件发生的话就要向应用程序发送按键值，首先判断按键值的最高位是否为 1，如果为 1 的话就表示按键值有效。如果按键值有效的话就将最高位清除，得到真实的按键值，然后通过 copy_to_user 函数返回给应用程序。向应用程序发送按键值完成以后就将 imx6uirq 结构体的 releasekey 成员变量清零，准备下一次按键操作。

第 210~214 行，按键中断驱动操作函数集 imx6uirq_fops。

第 221~253 行，驱动入口函数，第 250 和 251 行分别初始化 imx6uirq 结构体中的原子变量 keyvalue 和 releasekey，第 252 行调用 keyio_init 函数初始化按键所使用的 IO。

第 261~275 行，驱动出口函数，第 265 行调用 del_timer_sync 函数删除定时器，第 268~070 行轮流释放申请的所有按键中断。

51.3.3 编写测试 APP

测试 APP 要实现的内容很简单，通过不断的读取/dev/imx6uirq 文件来获取按键值，当按键按下以后就会将获取到的按键值输出在终端上，新建名为 imx6uirqApp.c 的文件，然后输入如下所示内容：

示例代码 51.3.3.1 imx6uirqApp.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  #include "linux/ioctl.h"
9  /***************************************************************
10 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
11 文件名   : imx6uirqApp.c
```

```
12 作者       : 左忠凯
13 版本       : V1.0
14 描述       : 定时器测试应用程序
15 其他       : 无
16 使用方法 : ./imx6uirqApp /dev/imx6uirq 打开测试 App
17 论坛       : www.openedv.com
18 日志       初版 V1.0 2019/7/26 左忠凯创建
19 ***************************************************************/
20
21 /*
22  * @description     : main 主程序
23  * @param - argc    : argv 数组元素个数
24  * @param - argv    : 具体参数
25  * @return          : 0 成功;其他 失败
26  */
27 int main(int argc, char *argv[])
28 {
29     int fd;
30     int ret = 0;
31     char *filename;
32
33     if (argc != 2) {
34         printf("Error Usage!\r\n");
35         return -1;
36     }
37
38     filename = argv[1];
39     fd = open(filename, O_RDWR);
40     if (fd < 0) {
41         printf("Can't open file %s\r\n", filename);
42         return -1;
43     }
44
45     while (1) {
46         ret = read(fd, &data, sizeof(data));
47         if (ret < 0) {    /* 数据读取错误或者无效  */
48
49         } else {          /* 数据读取正确         */
50             if (data)     /* 读取到数据           */
51                 printf("key value = %#X\r\n", data);
52         }
53  }
54  close(fd);
```

```
55  return ret;
56 }
```

第 45~53 行的 while 循环用于不断的读取按键值，如果读取到有效的按键值就将其输出到终端上。

51.4 运行测试

51.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 imx6uirq.o，Makefile 内容如下所示：

示例代码 51.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := imx6uirq.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 imx6uirq.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“imx6uirq.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 imx6uirqApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc imx6uirqApp.c -o imx6uirqApp
```

编译成功以后就会生成 imx6uirqApp 这个应用程序。

51.4.2 运行测试

将上一小节编译出来 imx6uirq.ko 和 imx6uirqApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 imx6uirq.ko 驱动模块：

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe imx6uirq.ko      //加载驱动
```

驱动加载成功以后可以通过查看/proc/interrupts 文件来检查一下对应的中断有没有被注册上，输入如下命令：

```
cat /proc/interrupts
```

结果如图 51.4.2.1 所示：

```
/lib/modules/4.1.15 # cat /proc/interrupts
           CPU0
 16:      44744       GPC  55 Level     i.MX Timer Tick
 18:        405       GPC  26 Level     2020000.serial
 19:          0       GPC  98 Level     sai
 20:          0       GPC  50 Level     2034000.asrc
 49:          0  gpio-mxc  18 Edge      KEY0
 50:          0  gpio-mxc  19 Edge      2190000.usdhc cd
171:          0  gpio-mxc   4 Edge      headphone detect
```

KEY0中断

图 51.4.2.1 proc/interrupts 文件内容

从图 51.4.2.1 可以看出 imx6uirq.c 驱动文件里面的 KEY0 中断已经存在了，触发方式为跳边沿(Edge)，中断号为 49。

接下来使用如下命令来测试中断：

```
./imx6uirqApp /dev/imx6uirq
```

按下开发板上的 KEY0 键，终端就会输出按键值，如图 51.4.2.2 所示：

```
/lib/modules/4.1.15 # ./imx6uirqApp /dev/imx6uirq
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
```

图 51.4.2.2 读取到的按键值

从图 51.4.2.2 可以看出，按键值获取成功，并且不会有按键抖动导致的误判发生，说明按键消抖工作正常。如果要卸载驱动的话输入如下命令即可：

```
rmmod imx6uirq.ko
```

第五十二章 Linux 阻塞和非阻塞 IO 实验

阻塞和非阻塞 IO 是 Linux 驱动开发里面很常见的两种设备访问模式，在编写驱动的时候一定要考虑到阻塞和非阻塞。本章我们就来学习一下阻塞和非阻塞 IO，以及如何在驱动程序中处理阻塞与非阻塞，如何在驱动程序使用等待队列和 poll 机制。

52.1 阻塞和非阻塞 IO

52.1.1 阻塞和非阻塞简介

这里的“IO”并不是我们学习 STM32 或者其他单片机的时候所说的“GPIO”(也就是引脚)。这里的 IO 指的是 Input/Output，也就是输入/输出，是应用程序对驱动设备的输入/输出操作。当应用程序对设备驱动进行操作的时候，如果不能获取到设备资源，那么阻塞式 IO 就会将应用程序对应的线程挂起，直到设备资源可以才做为止。对于非阻塞 IO，应用程序对应的线程不会挂起，它要么一直轮询等待，直到设备资源可以使用，要么就直接放弃。阻塞式 IO 如图 52.1.1.1 所示：

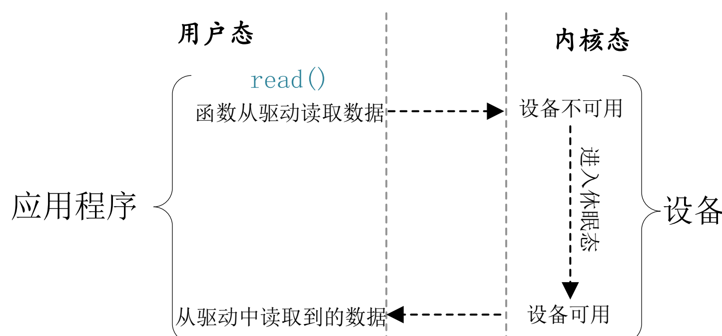


图 52.1.1.1 阻塞 IO 访问示意图。

图 52.1.1.1 中应用程序调用 read 函数从设备中读取数据，当设备不可用或数据未准备好的时候就会进入到休眠态。等设备可用的时候就会从休眠态唤醒，然后从设备中读取数据返回给应用程序。非阻塞 IO 如图 52.1.2 所示：

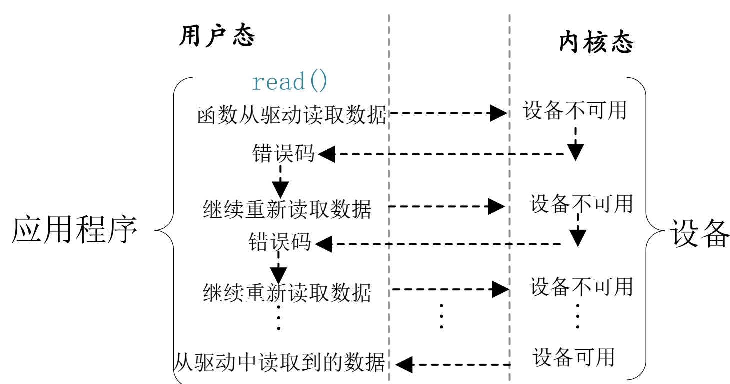


图 52.1.1.2 非阻塞 IO 访问示意图

从图 52.1.1.2 可以看出，应用程序使用非阻塞访问方式从设备读取数据，当设备不可用或数据未准备好的时候会立即向内核返回一个错误码，表示数据读取失败。应用程序会再次重新读取数据，这样一直往复循环，直到数据读取成功。

应用程序可以使用如下所示示例代码来实现阻塞访问：

示例代码 52.1.1.1 应用程序阻塞读取数据

```
1 int fd;
2 int data = 0;
3
4 fd = open("/dev/xxx_dev", O_RDWR);        /* 阻塞方式打开 */
5 ret = read(fd, &data, sizeof(data));    /* 读取数据     */
```

从示例代码 52.1.1.1 可以看出，对于设备驱动文件的默认读取方式就是阻塞式的，所以我们前面所有的例程测试 APP 都是采用阻塞 IO。

如果应用程序要采用非阻塞的方式来访问驱动设备文件，可以使用如下所示代码：

示例代码 52.1.1.2 应用程序非阻塞读取数据

```
1 int fd;
2 int data = 0;
3
4 fd = open("/dev/xxx_dev", O_RDWR | O_NONBLOCK);    /* 非阻塞方式打开 */
5 ret = read(fd, &data, sizeof(data));                /* 读取数据 */
```

第 4 行使用 open 函数打开"/dev/xxx_dev"设备文件的时候添加了参数"O_NONBLOCK"，表示以非阻塞方式打开设备，这样从设备中读取数据的时候就是非阻塞方式的了。

52.1.2 等待队列

1、等待队列头

阻塞访问最大的好处就是当设备文件不可操作的时候进程可以进入休眠态，这样可以将 CPU 资源让出来。但是，当设备文件可以操作的时候就必须唤醒进程，一般在中断函数里面完成唤醒工作。Linux 内核提供了等待队列(wait queue)来实现阻塞进程的唤醒工作，如果我们要在驱动中使用等待队列，必须创建并初始化一个等待队列头，等待队列头使用结构体 wait_queue_head_t 表示，wait_queue_head_t 结构体定义在文件 include/linux/wait.h 中，结构体内容如下所示：

示例代码 52.1.2.1 wait_queue_head_t 结构体

```
39 struct __wait_queue_head {
40  spinlock_t      lock;
41  struct list_head    task_list;
42 };
43 typedef struct __wait_queue_head wait_queue_head_t;
```

定义好等待队列头以后需要初始化，使数 init_waitqueue_head 函数初始化等待队列头，函数原型如下：

```
void init_waitqueue_head(wait_queue_head_t *q)
```

参数 q 就是要初始化的等待队列头。

也可以使用宏 DECLARE_WAIT_QUEUE_HEAD 来一次性完成等待队列头的定义的初始化。

2、等待队列项

等待队列头就是一个等待队列的头部，每个访问设备的进程都是一个队列项，当设备不可用的时候就要将这些进程对应的等待队列项添加到等待队列里面。结构体 wait_queue_t 表示等待队列项，结构体内容如下：

示例代码 52.1.2.2 wait_queue_t 结构体

```
struct __wait_queue {
    unsigned int           flags;
    void                   *private;
    wait_queue_func_t      func;
    struct list_head       task_list;
```

```
};
typedef struct __wait_queue wait_queue_t;
```

使用宏 DECLARE_WAITQUEUE 定义并初始化一个等待队列项，宏的内容如下：

```
DECLARE_WAITQUEUE(name, tsk)
```

name 就是等待队列项的名字，tsk 表示这个等待队列项属于哪个任务(进程)，一般设置为 current，在 Linux 内核中 current 相当于一个全局变量，表示当前进程。因此宏 DECLARE_WAITQUEUE 就是给当前正在运行的进程创建并初始化了一个等待队列项。

3、将队列项添加/移除等待队列头

当设备不可访问的时候就需要将进程对应的等待队列项添加到前面创建的等待队列头中，只有添加到等待队列头中以后进程才能进入休眠态。当设备可以访问以后再将进程对应的等待队列项从等待队列头中移除即可，等待队列项添加 API 函数如下：

```
void add_wait_queue(wait_queue_head_t    *q,
                    wait_queue_t         *wait)
```

函数参数和返回值含义如下：

q：等待队列项要加入的等待队列头。

wait：要加入的等待队列项。

返回值：无。

等待队列项移除 API 函数如下：

```
void remove_wait_queue(wait_queue_head_t    *q,
                       wait_queue_t         *wait)
```

函数参数和返回值含义如下：

q：要删除的等待队列项所处的等待队列头。

wait：要删除的等待队列项。

返回值：无。

4、等待唤醒

当设备可以使用的时候就要唤醒进入休眠态的进程，唤醒可以使用如下两个函数：

```
void wake_up(wait_queue_head_t *q)
void wake_up_interruptible(wait_queue_head_t *q)
```

参数 q 就是要唤醒的等待队列头，这两个函数会将这个等待队列头中的所有进程都唤醒。wake_up 函数可以唤醒处于 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 状态的进程，而 wake_up_interruptible 函数只能唤醒处于 TASK_INTERRUPTIBLE 状态的进程。

5、等待事件

除了主动唤醒以外，也可以设置等待队列等待某个事件，当这个事件满足以后就自动唤醒等待队列中的进程，和等待事件有关的 API 函数如表 52.1.2.1 所示：

函数	描述
wait_event(wq, condition)	等待以 wq 为等待队列头的等待队列被唤醒，前提是 condition 条件必须满足(为真)，否则一直阻塞。此函数会将进程设置为 TASK_UNINTERRUPTIBLE 状态
wait_event_timeout(wq, condition, timeout)	功能和 wait_event 类似，但是此函数可以添加超时时间，以 jiffies 为单位。此函数有返回值，如

	果返回 0 的话表示超时时间到，而且 condition 为假。为 1 的话表示 condition 为真，也就是条件满足了。
wait_event_interruptible(wq, condition)	与 wait_event 函数类似，但是此函数将进程设置为 TASK_INTERRUPTIBLE，就是可以被信号打断。
wait_event_interruptible_timeout(wq, condition, timeout)	与 wait_event_timeout 函数类似，此函数也将进程设置为 TASK_INTERRUPTIBLE，可以被信号打断。

表 52.1.2.1 等待事件 API 函数

52.1.3 轮询

如果用户应用程序以非阻塞的方式访问设备，设备驱动程序就要提供非阻塞的处理方式，也就是轮询。poll、epoll 和 select 可以用于处理轮询，应用程序通过 select、epoll 或 poll 函数来查询设备是否可以操作，如果可以操作的话就从设备读取或者向设备写入数据。当应用程序调用 select、epoll 或 poll 函数的时候设备驱动程序中的 poll 函数就会执行，因此需要在设备驱动程序中编写 poll 函数。我们先来看一下应用程序中使用的 select、poll 和 epoll 这三个函数。

1、select 函数

select 函数原型如下：

```
int select(int              nfds,
           fd_set           *readfds,
           fd_set           *writefds,
           fd_set           *exceptfds,
           struct timeval   *timeout)
```

函数参数和返回值含义如下：

nfds：要操作的文件描述符个数。

readfds、writefds 和 exceptfds：这三个指针指向描述符集合，这三个参数指明了关心哪些描述符、需要满足哪些条件等等，这三个参数都是 fd_set 类型的，fd_set 类型变量的每一个位都代表了一个文件描述符。readfds 用于监视指定描述符集的读变化，也就是监视这些文件是否可以读取，只要这些集合里面有一个文件可以读取那么 seclect 就会返回一个大于 0 的值表示文件可以读取。如果没有文件可以读取，那么就会根据 timeout 参数来判断是否超时。可以将 readfs 设置为 NULL，表示不关心任何文件的读变化。writefds 和 readfs 类似，只是 writefs 用于监视这些文件是否可以进行写操作。exceptfds 用于监视这些文件的异常。

比如我们现在要从一个设备文件中读取数据，那么就可以定义一个 fd_set 变量，这个变量要传递给参数 readfds。当我们定义好一个 fd_set 变量以后可以使用如下所示几个宏进行操作：

```
void FD_ZERO(fd_set *set)
void FD_SET(int fd, fd_set *set)
void FD_CLR(int fd, fd_set *set)
int   FD_ISSET(int fd, fd_set *set)
```

FD_ZERO 用于将 fd_set 变量的所有位都清零，FD_SET 用于将 fd_set 变量的某个位置 1，也就是向 fd_set 添加一个文件描述符，参数 fd 就是要加入的文件描述符。FD_CLR 用户将 fd_set 变量的某个位清零，也就是将一个文件描述符从 fd_set 中删除，参数 fd 就是要删除的文件描述

符。FD_ISSET 用于测试 fd_set 的某个位是否置 1，也就是判断某个文件是否可以进行操作，参数 fd 就是要判断的文件描述符。

timeout:超时时间，当我们调用 select 函数等待某些文件描述符可以设置超时时间，超时时间使用结构体 timeval 表示，结构体定义如下所示：

```
struct timeval {
    long    tv_sec;     /* 秒   */
    long    tv_usec;    /* 微妙 */
};
```

当 timeout 为 NULL 的时候就表示无限期的等待。

返回值：0，表示的话就表示超时发生，但是没有任何文件描述符可以进行操作；-1，发生错误；其他值，可以进行操作的文件描述符个数。

使用 select 函数对某个设备驱动文件进行读非阻塞访问的操作示例如下所示：

示例代码 52.1.3.1 select 函数非阻塞读访问示例

```
1  void main(void)
2  {
3      int ret, fd;                /* 要监视的文件描述符        */
4      fd_set readfds;             /* 读操作文件描述符集        */
5      struct timeval timeout;     /* 超时结构体                */
6  
7      fd = open("dev_xxx", O_RDWR | O_NONBLOCK);    /* 非阻塞式访问 */
8  
9      FD_ZERO(&readfds);          /* 清除 readfds              */
10     FD_SET(fd, &readfds);       /* 将 fd 添加到 readfds 里面 */
11 
12     /* 构造超时时间 */
13     timeout.tv_sec = 0;
14     timeout.tv_usec = 500000;   /* 500ms   */
15 
16     ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
17     switch (ret) {
18         case 0:                 /* 超时         */
19             printf("timeout!\r\n");
20             break;
21         case -1:                /* 错误         */
22             printf("error!\r\n");
23             break;
24         default:       /* 可以读取数据 */
25             if(FD_ISSET(fd, &readfds)) { /* 判断是否为 fd 文件描述符 */
26                 /* 使用 read 函数读取数据 */
27             }
28              break;
29     }
30 }
```

2、poll 函数

在单个线程中，select 函数能够监视的文件描述符数量有最大的限制，一般为 1024，可以修改内核将监视的文件描述符数量改大,但是这样会降低效率!这个时候就可以使用 poll 函数，poll 函数本质上和 select 没有太大的差别，但是 poll 函数没有最大文件描述符限制，Linux 应用程序中 poll 函数原型如下所示：

```
int poll(struct pollfd    *fds,
         nfds_t           nfds,
         int              timeout)
```

函数参数和返回值含义如下：

fds：要监视的文件描述符集合以及要监视的事件,为一个数组，数组元素都是结构体 pollfd 类型的，pollfd 结构体如下所示：

```
struct pollfd {
    int    fd;          /* 文件描述符  */
    short events;       /* 请求的事件  */
    short revents;      /* 返回的事件  */
};
```

fd 是要监视的文件描述符，如果 fd 无效的话那么 events 监视事件也就无效，并且 revents 返回 0。events 是要监视的事件，可监视的事件类型如下所示：

```
POLLIN          有数据可以读取。
POLLPRI         有紧急的数据需要读取。
POLLOUT         可以写数据。
POLLERR         指定的文件描述符发生错误。
POLLHUP         指定的文件描述符挂起。
POLLNVAL        无效的请求。
POLLRDNORM      等同于 POLLIN
```

revents 是返回参数，也就是返回的事件，有 Linux 内核设置具体的返回事件。

nfds：poll 函数要监视的文件描述符数量。

timeout：超时时间，单位为 ms。

返回值：返回 revents 域中不为 0 的 pollfd 结构体个数，也就是发生事件或错误的文件描述符数量；0，超时；-1，发生错误，并且设置 errno 为错误类型。

使用 poll 函数对某个设备驱动文件进行读非阻塞访问的操作示例如下所示：

示例代码 52.1.3.2 poll 函数读非阻塞访问示例

```
1  void main(void)
2  {
3      int ret;
4      int fd;                     /* 要监视的文件描述符      */
5      struct pollfd fds;
6
7      fd = open(filename, O_RDWR | O_NONBLOCK); /* 非阻塞式访问 */
8
9      /* 构造结构体   */
10     fds.fd = fd;
11     fds.events = POLLIN;    /* 监视数据是否可以读取 */
```

```
12
13     ret = poll(&fds, 1, 500);     /* 轮询文件是否可操作，超时 500ms */
14     if (ret) {                    /* 数据有效     */
15         ......
16         /* 读取数据 */
17         ......
18     } else if (ret == 0) {        /* 超时         */
19         ......
20     } else if (ret < 0) {         /* 错误         */
21         ......
22     }
23 }
```

3、epoll 函数

传统的 selcet 和 poll 函数都会随着所监听的 fd 数量的增加，出现效率低下的问题，而且 poll 函数每次必须遍历所有的描述符来检查就绪的描述符，这个过程很浪费时间。为此，epoll 因运而生，epoll 就是为处理大并发而准备的，一般常常在网络编程中使用 epoll 函数。应用程序需要先使用 epoll_create 函数创建一个 epoll 句柄，epoll_create 函数原型如下：

```
int epoll_create(int size)
```

函数参数和返回值含义如下：

size：从 Linux2.6.8 开始此参数已经没有意义了，随便填写一个大于 0 的值就可以。

返回值：epoll 句柄，如果为-1 的话表示创建失败。

epoll 句柄创建成功以后使用 epoll_ctl 函数向其中添加要监视的文件描述符以及监视的事件，epoll_ctl 函数原型如下所示：

```
int epoll_ctl(int                   epfd,
              int                   op,
              int                   fd,
              struct epoll_event    *event)
```

函数参数和返回值含义如下：

epfd：要操作的 epoll 句柄，也就是使用 epoll_create 函数创建的 epoll 句柄。

op：表示要对 epfd(epoll 句柄)进行的操作，可以设置为：

```
EPOLL_CTL_ADD      向 epfd 添加文件参数 fd 表示的描述符。
EPOLL_CTL_MOD      修改参数 fd 的 event 事件。
EPOLL_CTL_DEL      从 epfd 中删除 fd 描述符。
```

fd：要监视的文件描述符。

event：要监视的事件类型，为 epoll_event 结构体类型指针，epoll_event 结构体类型如下所示：

```
struct epoll_event {
    uint32_t       events;       /* epoll 事件      */
    epoll_data_t   data;         /*  用户数据       */
};
```

结构体 epoll_event 的 events 成员变量表示要监视的事件，可选的事件如下所示：

```
EPOLLIN         有数据可以读取。
EPOLLOUT        可以写数据。
```

EPOLLPRI 有紧急的数据需要读取。
EPOLLERR 指定的文件描述符发生错误。
EPOLLHUP 指定的文件描述符挂起。
EPOLLET 设置 epoll 为边沿触发，默认触发模式为水平触发。
EPOLLONESHOT 一次性的监视，当监视完成以后还需要再次监视某个 fd，那么就需要将 fd 重新添加到 epoll 里面。

上面这些事件可以进行“或”操作，也就是说可以设置监视多个事件。

返回值：0，成功；-1，失败，并且设置 errno 的值为相应的错误码。

一切都设置好以后应用程序就可以通过 epoll_wait 函数来等待事件的发生，类似 select 函数。epoll_wait 函数原型如下所示：

```
int epoll_wait(int                  epfd,
               struct epoll_event   *events,
               int                  maxevents,
               int                  timeout)
```

函数参数和返回值含义如下：

epfd：要等待的 epoll。

events：指向 epoll_event 结构体的数组，当有事件发生的时候 Linux 内核会填写 events，调用者可以根据 events 判断发生了哪些事件。

maxevents：events 数组大小，必须大于 0。

timeout：超时时间，单位为 ms。

返回值：0，超时；-1，错误；其他值，准备就绪的文件描述符数量。

epoll 更多的是用在大规模的并发服务器上，因为在这种场合下 select 和 poll 并不适合。当设计到的文件描述符(fd)比较少的时候就适合用 selcet 和 poll，本章我们就使用 sellect 和 poll 这两个函数。

52.1.4 Linux 驱动下的 poll 操作函数

当应用程序调用 select 或 poll 函数来对驱动程序进行非阻塞访问的时候，驱动程序 file_operations 操作集中的 poll 函数就会执行。所以驱动程序的编写者需要提供驱对应的 poll 函数，poll 函数原型如下所示：

```
unsigned int (*poll) (struct file *filp, struct poll_table_struct *wait)
```

函数参数和返回值含义如下：

filp：要打开的设备文件(文件描述符)。

wait：结构体 poll_table_struct 类型指针，又应用程序传递进来的。一般将此参数传递给 poll_wait 函数。

返回值:向应用程序返回设备或者资源状态，可以返回的资源状态如下：

POLLIN 有数据可以读取。
POLLPRI 有紧急的数据需要读取。
POLLOUT 可以写数据。
POLLERR 指定的文件描述符发生错误。
POLLHUP 指定的文件描述符挂起。
POLLNVAL 无效的请求。
POLLRDNORM 等同于 POLLIN，普通数据可读

我们需要在驱动程序的 poll 函数中调用 poll_wait 函数，poll_wait 函数不会引起阻塞，只是将应用程序添加到 poll_table 中，poll_wait 函数原型如下：

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
```

参数 wait_address 是要添加到 poll_table 中的等待队列头，参数 p 就是 poll_table，就是 file_operations 中 poll 函数的 wait 参数。

52.2 阻塞 IO 实验

在上一章 Linux 中断实验中，我们直接在应用程序中通过 read 函数不断的读取按键状态，当按键有效的时候就打印出按键值。这种方法有个缺点，那就是 imx6uirqApp 这个测试应用程序拥有很高的 CPU 占用率，大家可以在开发板中加载上一章的驱动程序模块 imx6uirq.ko，然后以后台运行模式打开 imx6uirqApp 这个测试软件，命令如下：

```
./imx6uirqApp /dev/imx6uirq &
```

测试驱动是否正常工作，如果驱动工作正常的话输入“top”命令查看 imx6uirqApp 这个应用程序的 CPU 使用率，结果如图 52.2.1 所示：

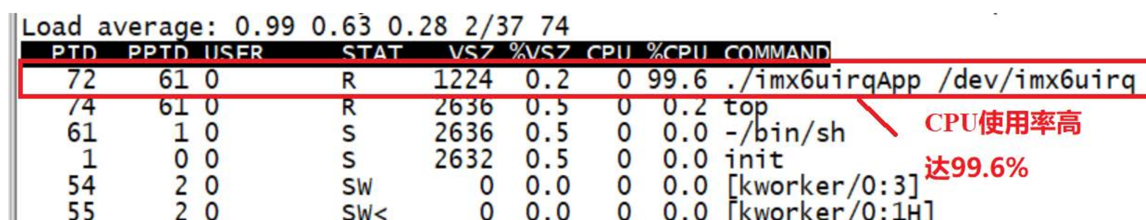


图 52.2.1 CPU 使用率

从图 52.2.1 可以看出，imx6uirqApp 这个应用程序的 CPU 使用率竟然高达 99.6%，这仅仅是一个读取按键值的应用程序，这么高的 CPU 使用率显然是有问题的！原因就在于我们是直接在 while 循环中通过 read 函数读取按键值，因此 imx6uirqApp 这个软件会一直运行，一直读取按键值，CPU 使用率肯定就会很高。最好的方法就是在没有有效的按键事件发生的时候，imx6uirqApp 这个应用程序应该处于休眠状态，当有按键事件发生以后 imx6uirqApp 这个应用程序才运行，打印出按键值，这样就会降低 CPU 使用率，本小节我们就使用阻塞 IO 来实现此功能。

52.2.1 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

52.2.2 实验程序编写

1、驱动程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 14_blockio。

本章实验我们在上一章的“13_irq”实验的基础上完成，主要是对其添加阻塞访问相关的代码。新建名为“14_blockio”的文件夹，然后在 14_blockio 文件夹里面创建 vscode 工程，工作区命名为“blockio”。将“13_irq”实验中的 imx6uirq.c 复制到 14_blockio 文件夹中，并重命名为 blockio.c。接下来我们就修改 blockio.c 这个文件，在其中添加阻塞相关的代码，完成以后的 blockio.c 内容如下所示(因为是在上一章实验的 imx6uirq.c 文件的基础上修改的，因为了减少篇幅，下面的代码有省略)：

示例代码 52.2.2.1 blockio.c 文件代码(有省略)

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
......
18  #include <asm/mach/map.h>
19  #include <asm/uaccess.h>
20  #include <asm/io.h>
21  /***************************************************************
22  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
23  文件名       : block.c
24  作者         : 左忠凯
25  版本         : V1.0
26  描述         : 阻塞 IO 访问
27  其他         : 无
28  论坛         : www.openedv.com
29  日志         : 初版 V1.0 2019/7/26 左忠凯创建
30  ***************************************************************/
31  #define IMX6UIRQ_CNT        1          /* 设备号个数       */
32  #define IMX6UIRQ_NAME       "blockio"  /* 名字             */
33  #define KEY0VALUE           0X01       /* KEY0 按键值      */
34  #define INVAKEY             0XFF       /* 无效的按键值     */
35  #define KEY_NUM             1          /* 按键数量         */
36
37  /* 中断 IO 描述结构体 */
38  struct irq_keydesc {
39      int gpio;                          /* gpio             */
40      int irqnum;                        /* 中断号           */
41      unsigned char value;               /* 按键对应的键值   */
42      char name[10];                     /* 名字             */
43      irqreturn_t (*handler)(int, void *);    /* 中断服务函数 */
44  };
45
46  /* imx6uirq 设备结构体 */
47  struct imx6uirq_dev{
48      dev_t devid;               /* 设备号                   */
49      struct cdev cdev;          /* cdev                     */
50      struct class *class;       /* 类                       */
51      struct device *device;     /* 设备                     */
52      int major;                 /* 主设备号                 */
53      int minor;                 /* 次设备号                 */
54      struct device_node  *nd; /* 设备节点                   */
55      atomic_t keyvalue;         /* 有效的按键键值           */
56      atomic_t releasekey;       /* 标记是否完成一次完成的按键 */
57      struct timer_list timer; /* 定义一个定时器*/
```

```
58      struct irq_keydesc irqkeydesc[KEY_NUM];    /* 按键 init 述数组  */
59      unsigned char curkeynum;                   /* 当前 init 按键号  */
60  
61      wait_queue_head_t r_wait;   /* 读等待队列头 */
62  };
63  
64  struct imx6uirq_dev imx6uirq;   /* irq 设备     */
65  
66  /* @description       : 中断服务函数，开启定时器
67   *                      定时器用于按键消抖。
68   * @param - irq       : 中断号
69   * @param - dev_id    : 设备结构。
70   * @return            : 中断执行结果
71   */
72  static irqreturn_t key0_handler(int irq, void *dev_id)
73  {
74      struct imx6uirq_dev *dev = (struct imx6uirq_dev*)dev_id;
75  
76      dev->curkeynum = 0;
77      dev->timer.data = (volatile long)dev_id;
78      mod_timer(&dev->timer, jiffies + msecs_to_jiffies(10));
79      return IRQ_RETVAL(IRQ_HANDLED);
80  }
81  
82  /* @description   : 定时器服务函数，用于按键消抖，定时器到了以后
83   *                  再次读取按键值，如果按键还是处于按下状态就表示按键有效。
84   * @param - arg   : 设备结构变量
85   * @return        : 无
86   */
87  void timer_function(unsigned long arg)
88  {
89      unsigned char value;
90      unsigned char num;
91      struct irq_keydesc *keydesc;
92      struct imx6uirq_dev *dev = (struct imx6uirq_dev *)arg;
93  
94      num = dev->curkeynum;
95      keydesc = &dev->irqkeydesc[num];
96  
97      value = gpio_get_value(keydesc->gpio);     /* 读取 IO 值    */
98      if(value == 0){                            /* 按下按键      */
99          atomic_set(&dev->keyvalue, keydesc->value);
100     }
```

```
101     else{                                       /* 按键松开     */
102         atomic_set(&dev->keyvalue, 0x80 | keydesc->value);
103         atomic_set(&dev->releasekey, 1);
104     }
105 
106     /* 唤醒进程 */
107     if(atomic_read(&dev->releasekey)) { /* 完成一次按键过程 */
108         /* wake_up(&dev->r_wait); */
109         wake_up_interruptible(&dev->r_wait);
110     }
111 }
112 
113 /*
114  * @description  : 按键 IO 初始化
115  * @param        : 无
116  * @return       : 无
117  */
118 static int keyio_init(void)
119 {
120     unsigned char i = 0;
121     char name[10];
122     int ret = 0;
......
163     /* 创建定时器 */
164     init_timer(&imx6uirq.timer);
165     imx6uirq.timer.function = timer_function;
166 
167     /* 初始化等待队列头 */
168     init_waitqueue_head(&imx6uirq.r_wait);
169     return 0;
170 }
171 
172 /*
173  * @description  : 打开设备
174  * @param - inode : 传递给驱动的 inode
175  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
176  *                 一般在 open 的时候将 private_data 指向设备结构体。
177  * @return        : 0 成功;其他 失败
178  */
179 static int imx6uirq_open(struct inode *inode, struct file *filp)
180 {
181     filp->private_data = &imx6uirq;    /* 设置私有数据 */
182     return 0;
```

```
183 }
184
185  /*
186   * @description      : 从设备读取数据
187   * @param - filp     : 要打开的设备文件(文件描述符)
188   * @param - buf      : 返回给用户空间的数据缓冲区
189   * @param - cnt      : 要读取的数据长度
190   * @param - offt     : 相对于文件首地址的偏移
191   * @return           : 读取的字节数，如果为负值，表示读取失败
192   */
193 static ssize_t imx6uirq_read(struct file *filp, char __user *buf,
                              size_t cnt, loff_t *offt)
194 {
195     int ret = 0;
196     unsigned char keyvalue = 0;
197     unsigned char releasekey = 0;
198     struct imx6uirq_dev *dev = (struct imx6uirq_dev *)
                                  filp->private_data;
199
200 #if 0
201     /* 加入等待队列，等待被唤醒,也就是有按键按下 */
202     ret = wait_event_interruptible(dev->r_wait,
                            atomic_read(&dev->releasekey));
203     if (ret) {
204         goto wait_error;
205     }
206 #endif
207
208     DECLARE_WAITQUEUE(wait, current);  /* 定义一个等待队列 */
209     if(atomic_read(&dev->releasekey) == 0) {  /* 没有按键按下 */
210         add_wait_queue(&dev->r_wait, &wait);  /* 添加到等待队列头 */
211         __set_current_state(TASK_INTERRUPTIBLE);/* 设置任务状态 */
212         schedule();                           /* 进行一次任务切换 */
213         if(signal_pending(current)) {   /* 判断是否为信号引起的唤醒 */
214             ret = -ERESTARTSYS;
215             goto wait_error;
216         }
217     }
218     remove_wait_queue(&dev->r_wait, &wait);/* 唤醒以后将等待队列移除 */
219
220     keyvalue = atomic_read(&dev->keyvalue);
221     releasekey = atomic_read(&dev->releasekey);
......
```

```
234     return 0;
235 
236 wait_error:
237     set_current_state(TASK_RUNNING);          /* 设置任务为运行态  */
238     remove_wait_queue(&dev->r_wait, &wait);   /* 将等待队列移除    */
239     return ret;
240 
241 data_error:
242     return -EINVAL;
243 }
244 
245 /* 设备操作函数 */
246 static struct file_operations imx6uirq_fops = {
247     .owner = THIS_MODULE,
248     .open = imx6uirq_open,
249     .read = imx6uirq_read,
250 };
251 
252 /*
253  * @description   : 驱动入口函数
254  * @param         : 无
255  * @return        : 无
256  */
257 static int __init imx6uirq_init(void)
258 {
259     /* 1、构建设备号 */
260     if (imx6uirq.major) {
261         imx6uirq.devid = MKDEV(imx6uirq.major, 0);
262         register_chrdev_region(imx6uirq.devid, IMX6UIRQ_CNT,
                                        IMX6UIRQ_NAME);
263     } else {
264         alloc_chrdev_region(&imx6uirq.devid, 0, IMX6UIRQ_CNT,
                                        IMX6UIRQ_NAME);
265         imx6uirq.major = MAJOR(imx6uirq.devid);
266         imx6uirq.minor = MINOR(imx6uirq.devid);
267     }
......
284 
285     /* 5、始化按键 */
286     atomic_set(&imx6uirq.keyvalue, INVAKEY);
287     atomic_set(&imx6uirq.releasekey, 0);
288     keyio_init();
289     return 0;
```

```
290 }
291
292 /*
293  * @description   : 驱动出口函数
294  * @param         : 无
295  * @return        : 无
296  */
297 static void __exit imx6uirq_exit(void)
298 {
299     unsigned i = 0;
300     /* 删除定时器 */
301     del_timer_sync(&imx6uirq.timer);    /* 删除定时器 */
......
310     class_destroy(imx6uirq.class);
311 }
312
313 module_init(imx6uirq_init);
314 module_exit(imx6uirq_exit);
315 MODULE_LICENSE("GPL");
```

第 32 行，修改设备文件名字为“blockio”，当驱动程序加载成功以后就会在根文件系统中出现一个名为“/dev/blockio”的文件。

第 61 行，在设备结构体中添加一个等待队列头 r_wait，因为在 Linux 驱动中处理阻塞 IO 需要用到等待队列。

第 107~110 行，定时器中断处理函数执行，表示有按键按下，先在 107 行判断一下是否是一次有效的按键，如果是的话就通过 wake_up 或者 wake_up_interruptible 函数来唤醒等待队列 r_wait。

第 168 行，调用 init_waitqueue_head 函数初始化等待队列头 r_wait。

第 200~206 行，采用等待事件来处理 read 的阻塞访问，wait_event_interruptible 函数等待 releasekey 有效，也就是有按键按下。如果按键没有按下的话进程就会进入休眠状态，因为采用了 wait_event_interruptible 函数，因此进入休眠态的进程可以被信号打断。

第 208~218 行，首先使用 DECLARE_WAITQUEUE 宏定义一个等待队列，如果没有按键按下的话就使用 add_wait_queue 函数将当前任务的等待队列添加到等待队列头 r_wait 中。随后调用__set_current_state 函数设置当前进程的状态为 TASK_INTERRUPTIBLE，也就是可以被信号打断。接下来调用 schedule 函数进行一次任务切换，当前进程就会进入到休眠态。如果有按键按下，那么进入休眠态的进程就会唤醒，然后接着从休眠点开始运行。在这里也就是从第 213 行开始运行，首先通过 signal_pending 函数判断一下进程是不是由信号唤醒的，如果是由信号唤醒的话就直接返回-ERESTARTSYS 这个错误码。如果不是由信号唤醒的(也就是被按键唤醒的)那么就在 218 行调用 remove_wait_queue 函数将进程从等待队列中删除。

使用等待队列实现阻塞访问重点注意两点：

①、将任务或者进程加入到等待队列头，

②、在合适的点唤醒等待队列，一般都是中断处理函数里面。

2、编写测试 APP

本节实验的测试 APP 直接使用第 51.3.3 小节所编写的 imx6uirqApp.c，将 imx6uirqApp.c 复制到本节实验文件夹下，并且重命名为 blockioApp.c，不需要修改任何内容。

52.2.3 运行测试

1、编译驱动程序和测试 APP

①、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 blockio.o，Makefile 内容如下所示：

示例代码 52.2.3.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := blockio.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 blockio.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“blockio.ko”的驱动模块文件。

②、编译测试 APP

输入如下命令编译测试 noblockioApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc blockioApp.c -o blockioApp
```

编译成功以后就会生成 blcokioApp 这个应用程序。

2、运行测试

将上一小节编译出来 blockio.ko 和 blockioApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 blockio.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe blockio.ko     //加载驱动
```

驱动加载成功以后使用如下命令打开 blockioApp 这个测试 APP，并且以后台模式运行：

```
./blockioApp /dev/blockio &
```

按下开发板上的 KEY0 按键，结果如图 52.2.3.1 所示：

```
/lib/modules/4.1.15 # ./blockioApp /dev/blockio &
/lib/modules/4.1.15 # key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
key value = 0X1
```

图 52.2.3.1 测试 APP 运行测试

当按下 KEY0 按键以后 blockioApp 这个测试 APP 就会打印出按键值。输入“top”命令，查看 blockioAPP 这个应用 APP 的 CPU 使用率，如图 52.2.3.2 所示：

```
Mem: 51212K used, 457716K free, 0K shrd, 0K buff, 5784K cached
CPU:  8.3% usr  8.3% sys  0.0% nic 83.3% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.00 0.01 0.02 1/38 89
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
   89    61 0        R     2632  0.5   0  8.3 top
   61     1 0        S     2636  0.5   0  0.0 -/bin/sh
    1     0 0        S     2632  0.5   0  0.0 init
   84    61 0        S     1224  0.2   0  0.0 ./blockioApp /dev/blockio
    7     2 0        SW       0  0.0   0  0.0 [rcu_preempt]
```

CPU使用率为0.0%

图 52.2.3.2 应用程序 CPU 使用率

从图 52.2.3.2 可以看出，当我们在按键驱动程序里面加入阻塞访问以后，blockioApp 这个应用程序的 CPU 使用率从图 52.2.1 中的 99.6%降低到了 0.0%。大家注意，这里的 0.0%并不是说 blockioApp 这个应用程序不使用 CPU 了，只是因为使用率太小了，CPU 使用率可能为 0.00001%，但是图 52.2.3.2 只能显示出小数点后一位，因此就显示成了 0.0%。

我们可以使用“kill”命令关闭后台运行的应用程序，比如我们关闭掉 blockioApp 这个后台运行的应用程序。首先输出“Ctrl+C”关闭 top 命令界面，进入到命令行模式。然后使用“ps”命令查看一下 blockioApp 这个应用程序的 PID，如图 52.2.3.3 所示：

```
/lib/modules/4.1.15 # ps
PID   USER     TIME   COMMAND
    1 0        0:01 init
        ......
  136 0        0:00 [kworker/0:1]
  137 0        0:00 [kworker/0:0]
  149 0        0:00 ./blockioApp /dev/blockio
  150 0        0:00 ps
```

blockioApp这个应用程序的PID为149

图 52.2.3.3 当前系统所有进程的 ID

从图图 52.2.3.3 可以看出，blockioApp 这个应用程序的 PID 为 149，使用“kill -9 PID”即可“杀死”指定 PID 的进程，比如我们现在要“杀死”PID 为 149 的 blockioApp 应用程序，可是使用如下命令：

```
kill -9 149
```

输入上述命令以后终端显示如图 52.2.3.4 所示：

```
/lib/modules/4.1.15 # kill -9 149
[1]+  Killed                     ./blockioApp /dev/blockio
/lib/modules/4.1.15 #
```

图 52.2.3.4 kill 命令输出结果

从图 52.2.3.4 可以看出，“./blockioApp /dev/blockio”这个应用程序已经被“杀掉”了，在此输入“ps”命令查看当前系统运行的进程，会发现 blockioApp 已经不见了。这个就是使用 kill 命令“杀掉”指定进程的方法。

52.3 非阻塞 IO 实验

52.3.1 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

52.3.2 实验程序编写

1、驱动程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 15_noblockio。

本章实验我们在 52.2 小节中的“14_blockio”实验的基础上完成，上一小节实验我们已经在驱动中添加了阻塞 IO 的代码，本小节我们继续完善驱动，加入非阻塞 IO 驱动代码。新建名为“15_noblockio”的文件夹，然后在 15_noblockio 文件夹里面创建 vscode 工程，工作区命名为“noblockio”。将“14_blockio”实验中的 blockio.c 复制到 15_noblockio 文件夹中，并重命名为 noblockio.c。接下来我们就修改 noblockio.c 这个文件，在其中添加非阻塞相关的代码，完成以后的 noblockio.c 内容如下所示(因为是在上一小节实验的 blockio.c 文件的基础上修改的，因为了减少篇幅，下面的代码有省略)：

示例代码 52.3.3.1 noblockio.c 文件(有省略)

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
......
18  #include <linux/wait.h>
19  #include <linux/poll.h>
20  #include <asm/mach/map.h>
21  #include <asm/uaccess.h>
22  #include <asm/io.h>
23  /***************************************************************
24  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
25  文件名      : noblock.c
26  作者        : 左忠凯
27  版本        : V1.0
28  描述        : 非阻塞 IO 访问
29  其他        : 无
30  论坛        : www.openedv.com
31  日志        : 初版 V1.0 2019/7/26 左忠凯创建
32  ***************************************************************/
31  #define IMX6UIRQ_CNT        1            /* 设备号个数    */
32  #define IMX6UIRQ_NAME       "noblockio"  /* 名字          */
......
187  /*
188   * @description     : 从设备读取数据
189   * @param - filp    : 要打开的设备文件(文件描述符)
190   * @param - buf     : 返回给用户空间的数据缓冲区
191   * @param - cnt     : 要读取的数据长度
192   * @param - offt    : 相对于文件首地址的偏移
193   * @return          : 读取的字节数，如果为负值，表示读取失败
194   */
195 static ssize_t imx6uirq_read(struct file *filp, char __user *buf,
                                 size_t cnt, loff_t *offt)
```

```
196 {
197     int ret = 0;
198     unsigned char keyvalue = 0;
199     unsigned char releasekey = 0;
200     struct imx6uirq_dev *dev = (struct imx6uirq_dev *)
                                        filp->private_data;
201 
202     if (filp->f_flags & O_NONBLOCK) {          /* 非阻塞访问   */
203         if(atomic_read(&dev->releasekey) == 0)  /* 没有按键按下 */
204             return -EAGAIN;
205     } else {                                    /* 阻塞访问     */
206         /* 加入等待队列，等待被唤醒,也就是有按键按下 */
207         ret = wait_event_interruptible(dev->r_wait,
                                   atomic_read(&dev->releasekey));
208         if (ret) {
209             goto wait_error;
210         }
211     }
......
229 wait_error:
230     return ret;
231 data_error:
232     return -EINVAL;
233 }
234 
235  /*
236   * @description     : poll 函数，用于处理非阻塞访问
237   * @param - filp    : 要打开的设备文件(文件描述符)
238   * @param - wait    : 等待列表(poll_table)
239   * @return          : 设备或者资源状态，
240   */
241 unsigned int imx6uirq_poll(struct file *filp,
                              struct poll_table_struct *wait)
242 {
243     unsigned int mask = 0;
244     struct imx6uirq_dev *dev = (struct imx6uirq_dev *)
                                        filp->private_data;
245 
246     poll_wait(filp, &dev->r_wait, wait);
247 
248     if(atomic_read(&dev->releasekey)) {    /* 按键按下     */
249         mask = POLLIN | POLLRDNORM;        /* 返回 PLLIN   */
250     }
```

```
251     return mask;
252 }
253 
254 /* 设备操作函数 */
255 static struct file_operations imx6uirq_fops = {
256     .owner = THIS_MODULE,
257     .open = imx6uirq_open,
258     .read = imx6uirq_read,
259     .poll = imx6uirq_poll,
260 };
261 
262 /*
263  * @description     : 驱动入口函数
264  * @param           : 无
265  * @return          : 无
266  */
267 static int __init imx6uirq_init(void)
268 {
......
298     keyio_init();
299     return 0;
300 }
301 
302 /*
303  * @description     : 驱动出口函数
304  * @param           : 无
305  * @return          : 无
306  */
307 static void __exit imx6uirq_exit(void)
308 {
309     unsigned i = 0;
310     /* 删除定时器 */
311     del_timer_sync(&imx6uirq.timer);    /* 删除定时器 */
......
320     class_destroy(imx6uirq.class);
321 }
322 
323 module_init(imx6uirq_init);
324 module_exit(imx6uirq_exit);
325 MODULE_LICENSE("GPL");
```

第 32 行，修改设备文件名字为“noblockio”，当驱动程序加载成功以后就会在根文件系统中出现一个名为“/dev/noblockio”的文件。

第 202~204 行，判断是否为非阻塞式读取访问，如果是的话就判断按键是否有效，也就是判断一下有没有按键按下，如果没有的话就返回-EAGAIN。

第 241~252 行，imx6uirq_poll 函数就是 file_operations 驱动操作集中的 poll 函数，当应用程序调用 select 或者 poll 函数的时候 imx6uirq_poll 函数就会执行。第 246 行调用 poll_wait 函数将等待队列头添加到 poll_table 中，第 248~250 行判断按键是否有效，如果按键有效的话就向应用程序返回 POLLIN 这个事件，表示有数据可以读取。

第 259 行，设置 file_operations 的 poll 成员变量为 imx6uirq_poll。

2、编写测试 APP

新建名为 noblockioApp.c 测试 APP 文件，然后在其中输入如下所示内容：

示例代码 52.3.3.2 noblockioApp.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  #include "poll.h"
9  #include "sys/select.h"
10 #include "sys/time.h"
11 #include "linux/ioctl.h"
12 /***************************************************************
13 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
14 文件名      : noblockApp.c
15 作者        : 左忠凯
16 版本        : V1.0
17 描述        : 非阻塞访问测试 APP
18 其他        : 无
19 使用方法    : ./blockApp /dev/blockio 打开测试 App
20 论坛        : www.openedv.com
21 日志        : 初版 V1.0 2019/9/8 左忠凯创建
22 ***************************************************************/
23 
24 /*
25  * @description     : main 主程序
26  * @param - argc    : argv 数组元素个数
27  * @param - argv    : 具体参数
28  * @return          : 0 成功;其他 失败
29  */
30 int main(int argc, char *argv[])
31 {
32     int fd;
33     int ret = 0;
```

```
34      char *filename;
35      struct pollfd fds;
36      fd_set readfds;
37      struct timeval timeout;
38      unsigned char data;
39
40      if (argc != 2) {
41          printf("Error Usage!\r\n");
42          return -1;
43      }
44
45      filename = argv[1];
46      fd = open(filename, O_RDWR | O_NONBLOCK);   /* 非阻塞访问 */
47      if (fd < 0) {
48          printf("Can't open file %s\r\n", filename);
49          return -1;
50      }
51
52  #if 0
53      /* 构造结构体 */
54      fds.fd = fd;
55      fds.events = POLLIN;
56
57      while (1) {
58          ret = poll(&fds, 1, 500);
59          if (ret) {  /* 数据有效 */
60              ret = read(fd, &data, sizeof(data));
61              if(ret < 0) {
62                  /* 读取错误 */
63              } else {
64                  if(data)
65                      printf("key value = %d \r\n", data);
66              }
67          } else if (ret == 0) {   /* 超时 */
68              /* 用户自定义超时处理 */
69          } else if (ret < 0) {    /* 错误 */
70              /* 用户自定义错误处理 */
71          }
72      }
73  #endif
74
75      while (1) {
76          FD_ZERO(&readfds);
```

```
77          FD_SET(fd, &readfds);
78          /* 构造超时时间 */
79          timeout.tv_sec = 0;
80          timeout.tv_usec = 500000; /* 500ms */
81          ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
82          switch (ret) {
83              case 0:     /* 超时 */
84                  /* 用户自定义超时处理 */
85                  break;
86              case -1:    /* 错误 */
87                  /* 用户自定义错误处理 */
88                  break;
89              default:  /* 可以读取数据 */
90                  if(FD_ISSET(fd, &readfds)) {
91                      ret = read(fd, &data, sizeof(data));
92                      if (ret < 0) {
93                          /* 读取错误 */
94                      } else {
95                          if (data)
96                              printf("key value=%d\r\n", data);
97                      }
98                  }
99                  break;
100         }
101     }
102
103     close(fd);
104     return ret;
105 }
```

第 52~73 行，这段代码使用 poll 函数来实现非阻塞访问，在 while 循环中使用 poll 函数不断的轮询，检查驱动程序是否有数据可以读取，如果可以读取的话就调用 read 函数读取按键数据。

第 75~101 行，这段代码使用 select 函数来实现非阻塞访问。

52.3.3 运行测试

1、编译驱动程序和测试 APP

①、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 noblockio.o，Makefile 内容如下所示：

示例代码 52.3.3.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
```

```
......
4  obj-m := noblockio.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 noblockio.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“noblockio.ko”的驱动模块文件。

②、编译测试 APP

输入如下命令编译测试 noblockioApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc noblockioApp.c -o noblockioApp
```

编译成功以后就会生成 noblcokioApp 这个应用程序。

2、运行测试

将上一小节编译出来 noblockio.ko 和 noblockioApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 blockio.ko 驱动模块：

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe noblockio.ko  //加载驱动
```

驱动加载成功以后使用如下命令打开 noblockioApp 这个测试 APP，并且以后台模式运行：

```
./noblockioApp /dev/noblockio &
```

按下开发板上的 KEY0 按键，结果如图 52.3.3.1 所示：

```
/lib/modules/4.1.15 # ./noblockioApp /dev/noblockio &
/lib/modules/4.1.15 # key value = 1
key value = 1
key value = 1
key value = 1
key value = 1
key value = 1
key value = 1
```

图 52.3.3.1 测试 APP 运行测试

当按下 KEY0 按键以后 noblockioApp 这个测试 APP 就会打印出按键值。输入“top”命令，查看 noblockioAPP 这个应用 APP 的 CPU 使用率，如图 52.3.3.2 所示：

```
Mem: 58080K used, 450848K free, 0K shrd, 0K buff, 5816K cached
CPU:  9.0% usr  0.0% sys  0.0% nic 90.9% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 0.00 0.01 0.05 1/38 135
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
   61     1 0        S     2636  0.5   0  0.0 -/bin/sh
    1     0 0        S     2632  0.5   0  0.0 init
  135    61 0        R     2632  0.5   0  0.0 top
  134    61 0        S     1224  0.2   0  0.0 ./noblockioApp /dev/noblockio
```

CPU使用率0.0%

图 52.3.3.2 应用程序 CPU 使用率

从图 52.3.3.2 可以看出，采用非阻塞方式读处理以后，noblockioApp 的 CPU 占用率也低至 0.0%，和图 52.2.3.2 中的 blockioApp 一样，这里的 0.0%并不是说 noblockioApp 这个应用程序不使用 CPU 了，只是因为使用率太小了，而图中只能显示出小数点后一位，因此就显示成了 0.0%。

如果要“杀掉”处于后台运行模式的 noblockioApp 这个应用程序，可以参考 52.2.3 小节讲解的方法。

第五十三章 异步通知实验

在前面使用阻塞或者非阻塞的方式来读取驱动中按键值都是应用程序主动读取的，对于非阻塞方式来说还需要应用程序通过 poll 函数不断的轮询。最好的方式就是驱动程序能主动向应用程序发出通知，报告自己可以访问，然后应用程序在从驱动程序中读取或写入数据，类似于我们在裸机例程中讲解的中断。Linux 提供了异步通知这个机制来完成此功能，本章我们就来学习一下异步通知以及如何在驱动中添加异步通知相关处理代码。

53.1 异步通知

53.1.1 异步通知简介

我们首先来回顾一下"中断"，中断是处理器提供的一种异步机制，我们配置好中断以后就可以让处理器去处理其他的事情了，当中断发生以后会触发我们事先设置好的中断服务函数，在中断服务函数中做具体的处理。比如我们在裸机篇里面编写的 GPIO 按键中断实验，我们通过按键去开关蜂鸣器，采用中断以后处理器就不需要时刻的去查看按键有没有被按下，因为按键按下以后会自动触发中断。同样的，Linux 应用程序可以通过阻塞或者非阻塞这两种方式来访问驱动设备，通过阻塞方式访问的话应用程序会处于休眠态，等待驱动设备可以使用，非阻塞方式的话会通过 poll 函数来不断的轮询，查看驱动设备文件是否可以使用。这两种方式都需要应用程序主动的去查询设备的使用情况，如果能提供一种类似中断的机制，当驱动程序可以访问的时候主动告诉应用程序那就最好了。

"信号"为此应运而生，信号类似于我们硬件上使用的"中断"，只不过信号是软件层次上的。算是在软件层次上对中断的一种模拟，驱动可以通过主动向应用程序发送信号的方式来报告自己可以访问了，应用程序获取到信号以后就可以从驱动设备中读取或者写入数据了。整个过程就相当于应用程序收到了驱动发送过来了的一个中断，然后应用程序去响应这个中断，在整个处理过程中应用程序并没有去查询驱动设备是否可以访问，一切都是由驱动设备自己告诉给应用程序的。

阻塞、非阻塞、异步通知，这三种是针对不同的场合提出来的不同的解决方法，没有优劣之分，在实际的工作和学习中，根据自己的实际需求选择合适的处理方法即可。

异步通知的核心就是信号，在 arch/xtensa/include/uapi/asm/signal.h 文件中定义了 Linux 所支持的所有信号，这些信号如下所示：

示例代码 53.1.1.1 Linux 信号

```
34 #define SIGHUP       1      /* 终端挂起或控制进程终止       */
35 #define SIGINT       2      /* 终端中断(Ctrl+C 组合键)      */
36 #define SIGQUIT      3      /* 终端退出(Ctrl+\组合键)       */
37 #define SIGILL       4      /* 非法指令                     */
38 #define SIGTRAP      5      /* debug 使用，有断点指令产生   */
39 #define SIGABRT      6      /* 由 abort(3)发出的退出指令    */
40 #define SIGIOT       6      /* IOT 指令                     */
41 #define SIGBUS       7      /* 总线错误                     */
42 #define SIGFPE       8      /* 浮点运算错误                 */
43 #define SIGKILL      9      /* 杀死、终止进程               */
44 #define SIGUSR1      10     /* 用户自定义信号 1             */
45 #define SIGSEGV      11     /* 段违例(无效的内存段)         */
46 #define SIGUSR2      12     /* 用户自定义信号 2             */
47 #define SIGPIPE      13     /* 向非读管道写入数据           */
48 #define SIGALRM      14     /* 闹钟                         */
49 #define SIGTERM      15     /* 软件终止                     */
50 #define SIGSTKFLT    16     /* 栈异常                       */
51 #define SIGCHLD      17     /* 子进程结束                   */
52 #define SIGCONT      18     /* 进程继续                     */
53 #define SIGSTOP      19     /* 停止进程的执行，只是暂停     */
```

```
54 #define SIGTSTP       20     /* 停止进程的运行(Ctrl+Z 组合键)    */
55 #define SIGTTIN       21     /* 后台进程需要从终端读取数据        */
56 #define SIGTTOU       22     /* 后台进程需要向终端写数据      */
57 #define SIGURG        23     /* 有"紧急"数据                  */
58 #define SIGXCPU       24     /* 超过 CPU 资源限制             */
59 #define SIGXFSZ       25     /* 文件大小超额                  */
60 #define SIGVTALRM     26     /* 虚拟时钟信号                  */
61 #define SIGPROF       27     /* 时钟信号描述                  */
62 #define SIGWINCH      28     /* 窗口大小改变                  */
63 #define SIGIO         29     /* 可以进行输入/输出操作         */
64 #define SIGPOLL       SIGIO
65 /* #define SIGLOS    29 */
66 #define SIGPWR        30     /* 断点重启                      */
67 #define SIGSYS        31     /* 非法的系统调用                */
68 #define  SIGUNUSED    31     /* 未使用信号                    */
```

在示例代码 53.1.1.1 中的这些信号中，除了 SIGKILL(9)和 SIGSTOP(19)这两个信号不能被忽略外，其他的信号都可以忽略。这些信号就相当于中断号，不同的中断号代表了不同的中断，不同的中断所做的处理不同，因此，驱动程序可以通过向应用程序发送不同的信号来实现不同的功能。

我们使用中断的时候需要设置中断处理函数，同样的，如果要在应用程序中使用信号，那么就必须设置信号所使用的信号处理函数，在应用程序中使用 signal 函数来设置指定信号的处理函数，signal 函数原型如下所示：

```
sighandler_t signal(int signum, sighandler_t handler)
```

函数参数和返回值含义如下：

signum：要设置处理函数的信号。

handler：信号的处理函数。

返回值：设置成功的话返回信号的前一个处理函数，设置失败的话返回 SIG_ERR。

信号处理函数原型如下所示：

```
typedef void (*sighandler_t)(int)
```

我们前面讲解的使用“kill -9 PID”杀死指定进程的方法就是向指定的进程(PID)发送 SIGKILL 这个信号。当按下键盘上的 CTRL+C 组合键以后会向当前正在占用终端的应用程序发出 SIGINT 信号，SIGINT 信号默认的动作是关闭当前应用程序。这里我们修改一下 SIGINT 信号的默认处理函数，当按下 CTRL+C 组合键以后先在终端上打印出“SIGINT signal!”这行字符串，然后再关闭当前应用程序。新建 signaltest.c 文件，然后输入如下所示内容：

示例代码 53.1.1.2 信号测试

```
1 #include "stdlib.h"
2 #include "stdio.h"
3 #include "signal.h"
4 
5 void sigint_handler(int num)
6 {
7     printf("\r\nSIGINT signal!\r\n");
8     exit(0);
```

```
9  }
10
11 int main(void)
12 {
13     signal(SIGINT, sigint_handler);
14     while(1);
15     return 0;
16 }
```

在示例代码 53.1.1.2 中我们设置 SIGINT 信号的处理函数为 sigint_handler，当按下 CTRL+C 向 signaltest 发送 SIGINT 信号以后 sigint_handler 函数就会执行，此函数先输出一行“SIGINT signal!”字符串，然后调用 exit 函数关闭 signaltest 应用程序。

使用如下命令编译 signaltest.c：

```
gcc signaltest.c -o signaltest
```

然后输入“./signaltest”命令打开 signaltest 这个应用程序，然后按下键盘上的 CTRL+C 组合键，结果如图 53.1.1.1 所示：

```
zuozhongkai@ubuntu:~$ ./signaltest
^C
SIGINT signal!
zuozhongkai@ubuntu:~$
```

图 53.1.1.1 signaltest 软件运行结果

从图 53.1.1.1 可以看出，当按下 CTRL+C 组合键以后 sigint_handler 这个 SIGINT 信号处理函数执行了，并且输出了“SIGINT signal!”这行字符串。

53.1.2 驱动中的信号处理

1、fasync_struct 结构体

首先我们需要在驱动程序中定义一个 fasync_struct 结构体指针变量，fasync_struct 结构体内容如下：

示例代码 53.1.2.1 fasync_struct 发结构体

```
struct fasync_struct {
    spinlock_t              fa_lock;
    int                     magic;
    int                     fa_fd;
    struct fasync_struct *fa_next;
    struct file             *fa_file;
    struct rcu_head         fa_rcu;
};
```

一般将 fasync_struct 结构体指针变量定义到设备结构体中，比如在上一章节的 imx6uirq_dev 结构体中添加一个 fasync_struct 结构体指针变量，结果如下所示：

示例代码 53.1.2.2 在设备结构体中添加 fasync_struct 类型变量指针

```
1  struct imx6uirq_dev {
2      struct device *dev;
3      struct class *cls;
```

```
4      struct cdev cdev;
......
14     struct fasync_struct *async_queue;  /* 异步相关结构体 */
15 };
```

第 14 行就是在 imx6uirq_dev 中添加了一个 fasync_struct 结构体指针变量。

2、fasync 函数

如果要使用异步通知，需要在设备驱动中实现 file_operations 操作集中的 fasync 函数，此函数格式如下所示：

```
int (*fasync) (int fd, struct file *filp, int on)
```

fasync 函数里面一般通过调用 fasync_helper 函数来初始化前面定义的 fasync_struct 结构体指针，fasync_helper 函数原型如下：

```
int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
```

fasync_helper 函数的前三个参数就是 fasync 函数的那三个参数，第四个参数就是要初始化的 fasync_struct 结构体指针变量。当应用程序通过“fcntl(fd, F_SETFL, flags | FASYNC)”改变 fasync 标记的时候，驱动程序 file_operations 操作集中的 fasync 函数就会执行。

驱动程序中的 fasync 函数参考示例如下：

示例代码 53.1.2.3 驱动中 fasync 函数参考示例

```
1  struct xxx_dev {
2      ......
3      struct fasync_struct *async_queue;  /* 异步相关结构体 */
4  };
5
6  static int xxx_fasync(int fd, struct file *filp, int on)
7  {
8      struct xxx_dev *dev = (xxx_dev)filp->private_data;
9
10     if (fasync_helper(fd, filp, on, &dev->async_queue) < 0)
11         return -EIO;
12     return 0;
13 }
14
15 static struct file_operations xxx_ops = {
16     ......
17     .fasync = xxx_fasync,
18     ......
19 };
```

在关闭驱动文件的时候需要在 file_operations 操作集中的 release 函数中释放 fasync_struct，fasync_struct 的释放函数同样为 fasync_helper，release 函数参数参考实例如下：

示例代码 53.1.2.4 释放 fasync_struct 参考示例

```
1 static int xxx_release(struct inode *inode, struct file *filp)
2 {
3     return xxx_fasync(-1, filp, 0); /* 删除异步通知 */
4 }
```

```
5
6 static struct file_operations xxx_ops = {
7    ......
8       .release = xxx_release,
9 };
```

第 3 行通过调用示例代码 53.1.2.3 中的 xxx_fasync 函数来完成 fasync_struct 的释放工作，但是，其最终还是通过 fasync_helper 函数完成释放工作。

1、kill_fasync 函数

当设备可以访问的时候，驱动程序需要向应用程序发出信号，相当于产生"中断"。kill_fasync 函数负责发送指定的信号，kill_fasync 函数原型如下所示：

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
```

函数参数和返回值含义如下：

fp： 要操作的 fasync_struct。

sig： 要发送的信号。

band： 可读时设置为 POLL_IN，可写时设置为 POLL_OUT。

返回值： 无。

53.1.3 应用程序对异步通知的处理

应用程序对异步通知的处理包括以下三部：

1、注册信号处理函数

应用程序根据驱动程序所使用的信号来设置信号的处理函数，应用程序使用 signal 函数来设置信号的处理函数。前面已经详细的讲过了，这里就不细讲了。

2、将本应用程序的进程号告诉给内核

使用 fcntl(fd, F_SETOWN, getpid())将本应用程序的进程号告诉给内核。

3、开启异步通知

使用如下两行程序开启异步通知：

```
flags = fcntl(fd, F_GETFL);              /*  获取当前的进程状态          */
fcntl(fd, F_SETFL, flags | FASYNC);      /*  开启当前进程异步通知功能    */
```

重点就是通过 fcntl 函数设置进程状态为 FASYNC，经过这一步，驱动程序中的 fasync 函数就会执行。

53.2 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

53.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 16_asyncnoti。

本章实验我们在上一章实验"15_noblockio"的基础上完成，在其中加入异步通知相关内容即可，当按键按下以后驱动程序向应用程序发送 SIGIO 信号，应用程序获取到 SIGIO 信号以后读取并且打印出按键值。

53.3.1 修改设备树文件

因为是在实验“15_noblockio”的基础上完成的，因此，不需要修改设备树。

53.3.2 程序编写

新建名为“16_asyncnoti”的文件夹，然后在 16_asyncnoti 文件夹里面创建 vscode 工程，工作区命名为“asyncnoti”。将“15_noblockio”实验中的 noblockio.c 复制到 16_asyncnoti 文件夹中，并重命名为 asyncnoti.c。接下来我们就修改 asyncnoti.c 这个文件，在其中添加异步通知关的代码，完成以后的 asyncnoti.c 内容如下所示(因为是在上一章实验的 noblockio.c 文件的基础上修改的，因为了减少篇幅，下面的代码有省略)：

示例代码 53.3.2.1 asyncnoti.c 文件代码段

```
1   #include <linux/types.h>
......
20  #include <linux/fcntl.h>
21  #include <asm/mach/map.h>
22  #include <asm/uaccess.h>
23  #include <asm/io.h>
24  /***************************************************************
25  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
26  文件名   : asyncnoti.c
27  作者     : 左忠凯
28  版本     : V1.0
29  描述     : 非阻塞 IO 访问
30  其他     : 无
31  论坛     : www.openedv.com
32  日志     : 初版 V1.0 2019/8/13 左忠凯创建
33  ***************************************************************/
34  #define IMX6UIRQ_CNT     1              /* 设备号个数      */
35  #define IMX6UIRQ_NAME    "asyncnoti"    /* 名字           */
36  #define KEY0VALUE        0X01           /* KEY0 按键值     */
37  #define INVAKEY          0XFF           /* 无效的按键值     */
38  #define KEY_NUM          1              /* 按键数量        */
......
49  /* imx6uirq 设备结构体 */
50  struct imx6uirq_dev{
......
64      struct fasync_struct *async_queue;     /* 异步相关结构体 */
65  };
66
67  struct imx6uirq_dev imx6uirq;   /* irq 设备 */
68
......
84
```

```
85  /* @description  : 定时器服务函数，用于按键消抖，定时器到了以后
86   *                再次读取按键值，如果按键还是处于按下状态就表示按键有效。
87   * @param - arg   : 设备结构变量
88   * @return        : 无
89   */
90  void timer_function(unsigned long arg)
91  {
92      unsigned char value;
93      unsigned char num;
94      struct irq_keydesc *keydesc;
95      struct imx6uirq_dev *dev = (struct imx6uirq_dev *)arg;
......
109     if(atomic_read(&dev->releasekey)) {    /* 一次完整的按键过程 */
110         if(dev->async_queue)
111             kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
112     }
113 
114 #if 0
115     /* 唤醒进程 */
116     if(atomic_read(&dev->releasekey)) {   /* 完成一次按键过程  */
117         /* wake_up(&dev->r_wait); */
118         wake_up_interruptible(&dev->r_wait);
119     }
120 #endif
121 }
......
262 /*
263  * @description   : fasync 函数，用于处理异步通知
264  * @param - fd    : 文件描述符
265  * @param - filp  : 要打开的设备文件(文件描述符)
266  * @param - on    : 模式
267  * @return        : 负数表示函数执行失败
268  */
269 static int imx6uirq_fasync(int fd, struct file *filp, int on)
270 {
271     struct imx6uirq_dev *dev = (struct imx6uirq_dev *)
                                          filp->private_data;
272     return fasync_helper(fd, filp, on, &dev->async_queue);
273 }
274 
275 /*
276  * @description   : release 函数，应用程序调用 close 关闭驱动文件的时候会执行
277  * @param – inode : inode 节点
```

```
278  * @param - filp : 要打开的设备文件(文件描述符)
279  * @return       : 负数表示函数执行失败
280  */
281 static int imx6uirq_release(struct inode *inode, struct file *filp)
282 {
283     return imx6uirq_fasync(-1, filp, 0);
284 }
285 
286 /* 设备操作函数 */
287 static struct file_operations imx6uirq_fops = {
288     .owner = THIS_MODULE,
289     .open = imx6uirq_open,
290     .read = imx6uirq_read,
291     .poll = imx6uirq_poll,
292     .fasync = imx6uirq_fasync,
293     .release = imx6uirq_release,
294 };
295 
296 /*
297  * @description   : 驱动入口函数
298  * @param         : 无
299  * @return        : 无
300  */
301 static int __init imx6uirq_init(void)
302 {
......
328 
329     /* 5、始化按键 */
330     atomic_set(&imx6uirq.keyvalue, INVAKEY);
331     atomic_set(&imx6uirq.releasekey, 0);
332     keyio_init();
333     return 0;
334 }
335 
336 /*
337  * @description   : 驱动出口函数
338  * @param         : 无
339  * @return        : 无
340  */
341 static void __exit imx6uirq_exit(void)
342 {
343     unsigned i = 0;
......
```

```
354     class_destroy(imx6uirq.class);
355 }
356 
357 module_init(imx6uirq_init);
358 module_exit(imx6uirq_exit);
359 MODULE_LICENSE("GPL");
```

第 20 行，添加 fcntl.h 头文件，因为要用到相关的 API 函数。

第 64 行，在设备结构体 imx6uirq_dev 中添加 fasync_struct 指针变量。

第 109~112 行，如果是一次完整的按键过程，那么就通过 kill_fasync 函数发送 SIGIO 信号。

第 114~120 行，屏蔽掉以前的唤醒进程相关程序。

第 269~273 行，imx6uirq_fasync 函数，为 file_operations 操作集中的 fasync 函数，此函数内容很简单，就是调用一下 fasync_helper。

第 281~284 行，release 函数，应用程序调用 close 函数关闭驱动设备文件的时候此函数就会执行，在此函数中释放掉 fasync_struct 指针变量。

第 292~293 行，设置 file_operations 操作集中的 fasync 和 release 这两个成员变量。

53.3.3 编写测试 APP

测试 APP 要实现的内容很简单，设置 SIGIO 信号的处理函数为 sigio_signal_func，当驱动程序向应用程序发送 SIGIO 信号以后 sigio_signal_func 函数就会执行。sigio_signal_func 函数内容很简单，就是通过 read 函数读取按键值。新建名为 asyncnotiApp.c 的文件，然后输入如下所示内容：

示例代码 53.3.3.2 asyncnotiApp.c 文件代码段

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  #include "poll.h"
9  #include "sys/select.h"
10 #include "sys/time.h"
11 #include "linux/ioctl.h"
12 #include "signal.h"
13 /***************************************************************
14 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
15 文件名       : asyncnotiApp.c
16 作者         : 左忠凯
17 版本         : V1.0
18 描述         : 异步通知测试 APP
19 其他         : 无
20 使用方法     : ./asyncnotiApp /dev/asyncnoti 打开测试 App
```

```
21 论坛          : www.openedv.com
22 日志          : 初版 V1.0 2019/8/13 左忠凯创建
23 ***************************************************************/
24 
25 static int fd = 0;   /* 文件描述符 */
26 
27 /*
28  * SIGIO 信号处理函数
29  * @param - signum     : 信号值
30  * @return             : 无
31  */
32 static void sigio_signal_func(int signum)
33 {
34     int err = 0;
35     unsigned int keyvalue = 0;
36 
37     err = read(fd, &keyvalue, sizeof(keyvalue));
38     if(err < 0) {
39         /* 读取错误 */
40     } else {
41         printf("sigio signal! key value=%d\r\n", keyvalue);
42     }
43 }
44 
45 /*
46  * @description    : main 主程序
47  * @param - argc   : argv 数组元素个数
48  * @param - argv   : 具体参数
49  * @return         : 0 成功;其他 失败
50  */
51 int main(int argc, char *argv[])
52 {
53     int flags = 0;
54     char *filename;
55 
56     if (argc != 2) {
57         printf("Error Usage!\r\n");
58         return -1;
59     }
60 
61     filename = argv[1];
62     fd = open(filename, O_RDWR);
63     if (fd < 0) {
```

```
64          printf("Can't open file %s\r\n", filename);
65          return -1;
66      }
67
68      /* 设置信号 SIGIO 的处理函数 */
69      signal(SIGIO, sigio_signal_func);
70
71      fcntl(fd, F_SETOWN, getpid()); /* 将当前进程的进程号告诉给内核  */
72      flags = fcntl(fd, F_GETFD);     /* 获取当前的进程状态            */
73      fcntl(fd, F_SETFL, flags | FASYNC);/* 设置进程启用异步通知功能  */
74
75      while(1) {
76          sleep(2);
77      }
78
79      close(fd);
80      return 0;
81  }
```

第 32~43 行，sigio_signal_func 函数，SIGIO 信号的处理函数，当驱动程序有效按键按下以后就会发送 SIGIO 信号，此函数就会执行。此函数通过 read 函数读取按键值，然后通过 printf 函数打印在终端上。

第 69 行，通过 signal 函数设置 SIGIO 信号的处理函数为 sigio_signal_func。

第 71~73 行，设置当前进程的状态，开启异步通知的功能。

第 75~77 行，while 循环，等待信号产生。

53.4 运行测试

53.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 asyncnoti.o，Makefile 内容如下所示：

示例代码 53.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := asyncnoti.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 asyncnoti.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“asyncnoti.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 asyncnotiApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc asyncnotiApp.c -o asyncnotiApp
```

编译成功以后就会生成 asyncnotiApp 这个应用程序。

53.4.2 运行测试

将上一小节编译出来 asyncnoti.ko 和 asyncnotiApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 asyncnoti.ko 驱动模块：

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe asyncnoti.ko  //加载驱动
```

驱动加载成功以后使用如下命令来测试中断：

```
./asyncnotiApp /dev/asyncnoti
```

按下开发板上的 KEY0 键，终端就会输出按键值，如图 53.4.2.1 所示：

```
/lib/modules/4.1.15 # ./asyncnotiApp /dev/asyncnoti

sigio signal! key value=1
sigio signal! key value=1
sigio signal! key value=1
sigio signal! key value=1
sigio signal! key value=1
sigio signal! key value=1
sigio signal! key value=1
```

图 53.4.2.1 读取到的按键值

从图 53.4.2.1 可以看出，捕获到 SIGIO 信号，并且按键值获取成功，大家可以自行以后台模式运行 asyncnotiApp，查看一下这个应用程序的 CPU 使用率。如果要卸载驱动的话输入如下命令即可：

```
rmmod asyncnoti.ko
```

第五十四章 platform 设备驱动实验

我们在前面几章编写的设备驱动都非常的简单，都是对 IO 进行最简单的读写操作。像 I2C、SPI、LCD 等这些复杂外设的驱动就不能这么去写了，Linux 系统要考虑到驱动的可重用性，因此提出了驱动的分离与分层这样的软件思路，在这个思路下诞生了我们将来最常打交道的 platform 设备驱动，也叫做平台设备驱动。本章我们就来学习一下 Linux 下的驱动分离与分层，以及 plartorm 框架下的设备驱动该如何不编写。

54.1 Linux 驱动的分离与分层

54.1.1 驱动的分隔与分离

对于 Linux 这样一个成熟、庞大、复杂的操作系统，代码的重用性非常重要，否则的话就会在 Linux 内核中存在大量无意义的重复代码。尤其是驱动程序，因为驱动程序占用了 Linux 内核代码量的大头，如果不对驱动程序加以管理，任由重复的代码肆意增加，那么用不了多久 Linux 内核的文件数量就庞大到无法接受的地步。

假如现在有三个平台 A、B 和 C，这三个平台(这里的平台说的是 SOC)上都有 MPU6050 这个 I2C 接口的六轴传感器，按照我们写裸机 I2C 驱动的时候的思路，每个平台都有一个 MPU6050 的驱动，因此编写出来的最简单的驱动框架如图 54.1.1 所示：

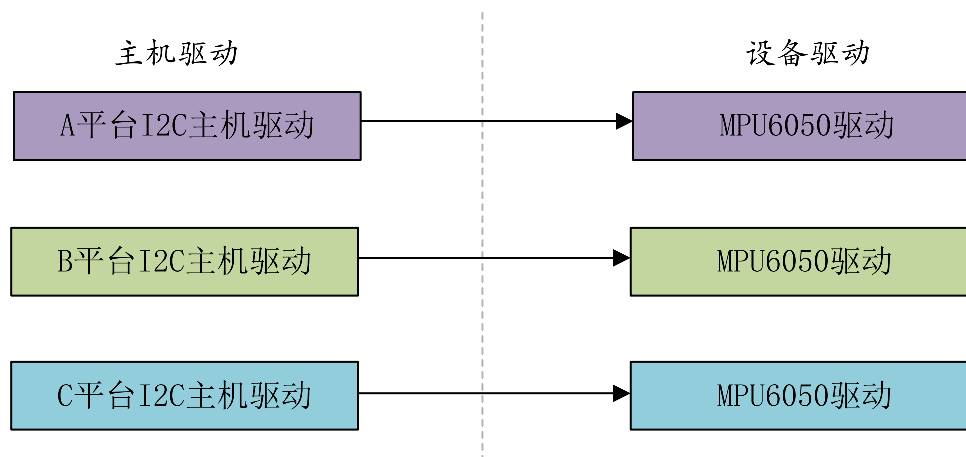


图 54.1.1 传统的 I2C 设备驱动

从图 54.1.1 可以看出，每种平台下都有一个主机驱动和设备驱动，主机驱动肯定是必须要的，毕竟不同的平台其 I2C 控制器不同。但是右侧的设备驱动就没必要每个平台都写一个，因为不管对于那个 SOC 来说，MPU6050 都是一样，通过 I2C 接口读取数据就行了，只需要一个 MPU6050 的驱动程序即可。如果在来几个 I2C 设备，比如 AT24C02、FT5206(电容触摸屏)等，如果按照图 54.1.1 中的写法，那么设备端的驱动将会重复的编写好几次。显然在 Linux 驱动程序中这种写法是不推荐的，最好的做法就是每个平台的 I2C 控制器都提供一个统一的接口(也叫做主机驱动)，每个设备的话也只提供一个驱动程序(设备驱动)，每个设备通过统一的 I2C 接口驱动来访问，这样就可以大大简化驱动文件，比如 54.1.1 中三种平台下的 MPU6050 驱动框架就可以简化为图 54.1.2 所示：

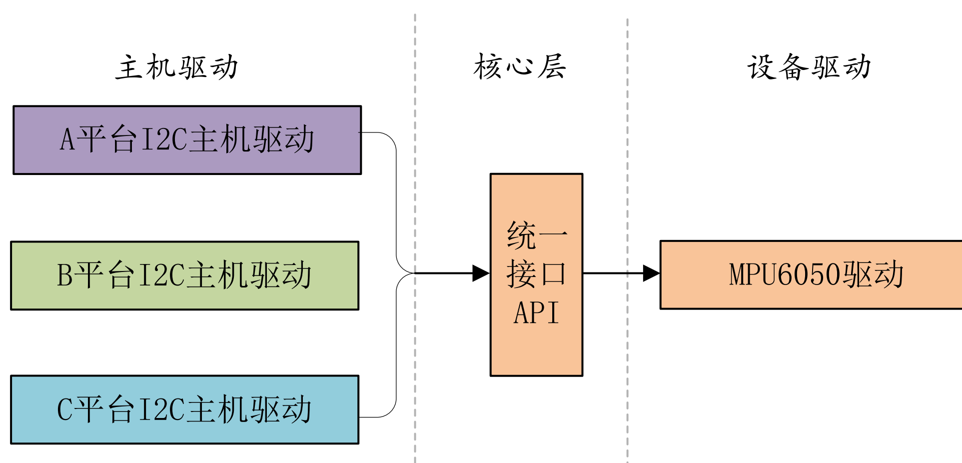


图 54.1.2 改进后的设备驱动

实际的 I2C 驱动设备肯定有很多种，不止 MPU6050 这一个，那么实际的驱动架构如图 54.1.3 所示：

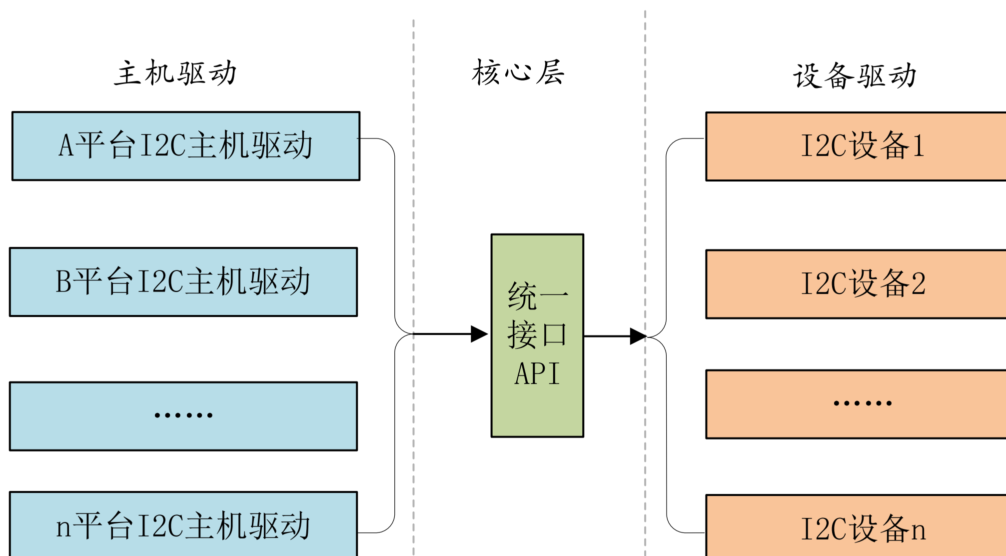


图 54.1.3 分隔后的驱动框架

这个就是驱动的分隔，也就是将主机驱动和设备驱动分隔开来，比如 I2C、SPI 等等都会采用驱动分隔的方式来简化驱动的开发。在实际的驱动开发中，一般 I2C 主机控制器驱动已经由半导体厂家编写好了，而设备驱动一般也有设备器件的厂家编写好了，我们只需要提供设备信息即可，比如 I2C 设备的话提供设备连接到了哪个 I2C 接口上，I2C 的速度是多少等等。相当于将设备信息从设备驱动中剥离开来，驱动使用标准方法去获取到设备信息(比如从设备树中获取到设备信息)，然后根据获取到的设备信息来初始化设备。 这样就相当于驱动只需要负责驱动，设备只需要设备，想办法将两者进行匹配即可。这个就是 Linux 中的总线(bus)、驱动(driver)和设备(device)模型，也就是常说的驱动分离。总线就是驱动和设备信息的月老，负责给两者牵线搭桥，如图 54.1.4 所示：

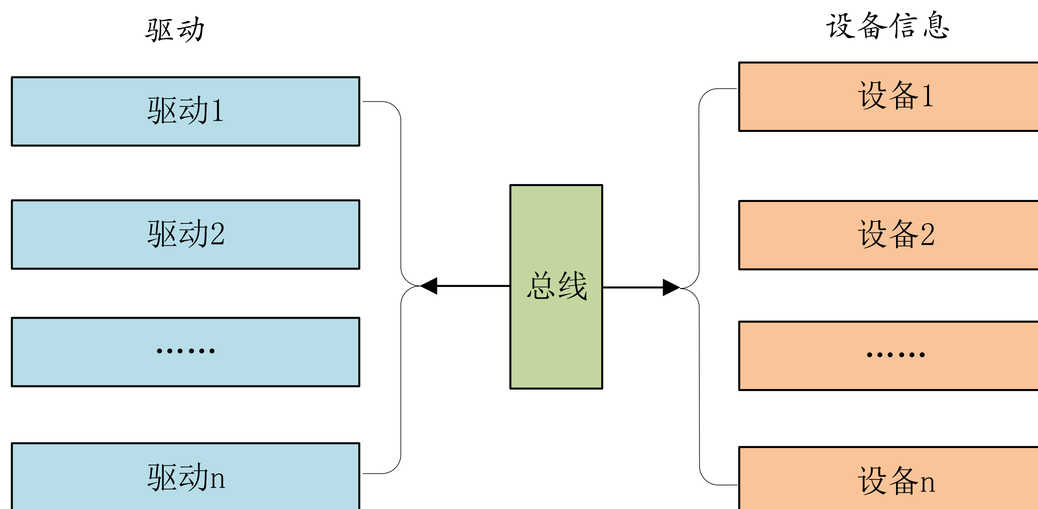


图 54.1.4 Linux 总线、驱动和设备模式

当我们向系统注册一个驱动的时候，总线就会在右侧的设备中查找，看看有没有与之匹配的设备，如果有的话就将两者联系起来。同样的，当向系统中注册一个设备的时候，总线就会在左侧的驱动中查找看有没有与之匹配的设备，有的话也联系起来。Linux 内核中大量的驱动程序都采用总线、驱动和设备模式，我们一会要重点讲解的 platform 驱动就是这一思想下的产物。

54.1.2 驱动的分层

上一小节讲了驱动的分隔与分离，本节我们来简单看一下驱动的分层，大家应该听说过网络的 7 层模型，不同的层负责不同的内容。同样的，Linux 下的驱动往往也是分层的，分层的目的也是为了在不同的层处理不同的内容。以其他书籍或者资料常常使用到的 input(输入子系统，后面会有专门的章节详细的讲解)为例，简单介绍一下驱动的分层。input 子系统负责管理所有跟输入有关的驱动，包括键盘、鼠标、触摸等，最底层的就是设备原始驱动，负责获取输入设备的原始值，获取到的输入事件上报给 input 核心层。input 核心层会处理各种 IO 模型，并且提供 file_operations 操作集合。我们在编写输入设备驱动的时候只需要处理好输入事件的上报即可，至于如何处理这些上报的输入事件那是上层去考虑的，我们不用管。可以看出借助分层模型可以极大的简化我们的驱动编写，对于驱动编写来说非常的友好。

54.2 platform 平台驱动模型简介

前面我们讲了设备驱动的分离，并且引出了总线(bus)、驱动(driver)和设备(device)模型，比如 I2C、SPI、USB 等总线。但是在 SOC 中有些外设是没有总线这个概念的，但是又要使用总线、驱动和设备模型该怎么办呢？为了解决此问题，Linux 提出了 platform 这个虚拟总线，相应的就有 platform_driver 和 platform_device。

54.2.1 platform 总线

Linux 系统内核使用 bus_type 结构体表示总线，此结构体定义在文件 include/linux/device.h，bus_type 结构体内容如下：

示例代码 54.2.1.1 bus_type 结构体代码段

```
1  struct bus_type {
```

```
2       const char      *name;                              /* 总线名字 */
3       const char      *dev_name;
4       struct device       *dev_root;
5       struct device_attribute *dev_attrs;
6       const struct attribute_group **bus_groups;      /* 总线属性 */
7       const struct attribute_group **dev_groups;      /* 设备属性 */
8       const struct attribute_group **drv_groups;      /* 驱动属性 */
9
10      int (*match)(struct device *dev, struct device_driver *drv);
11      int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
12      int (*probe)(struct device *dev);
13      int (*remove)(struct device *dev);
14      void (*shutdown)(struct device *dev);
15
16      int (*online)(struct device *dev);
17      int (*offline)(struct device *dev);
18      int (*suspend)(struct device *dev, pm_message_t state);
19      int (*resume)(struct device *dev);
20      const struct dev_pm_ops *pm;
21      const struct iommu_ops *iommu_ops;
22      struct subsys_private *p;
23      struct lock_class_key lock_key;
24 };
```

第 10 行，match 函数，此函数很重要，单词 match 的意思就是“匹配、相配”，因此此函数就是完成设备和驱动之间匹配的，总线就是使用 match 函数来根据注册的设备来查找对应的驱动，或者根据注册的驱动来查找相应的设备，因此每一条总线都必须实现此函数。match 函数有两个参数：dev 和 drv，这两个参数分别为 device 和 device_driver 类型，也就是设备和驱动。

platform 总线是 bus_type 的一个具体实例，定义在文件 drivers/base/platform.c，platform 总线定义如下：

示例代码 54.2.1.2 platform 总线实例

```
1 struct bus_type platform_bus_type = {
2       .name           = "platform",
3       .dev_groups     = platform_dev_groups,
4       .match          = platform_match,
5       .uevent         = platform_uevent,
6       .pm             = &platform_dev_pm_ops,
7 };
```

platform_bus_type 就是 platform 平台总线，其中 platform_match 就是匹配函数。我们来看一下驱动和设备是如何匹配的，platform_match 函数定义在文件 drivers/base/platform.c 中，函数内容如下所示：

示例代码 54.2.1.3 platform 总线实例

```
1  static int platform_match(struct device *dev,
                          struct device_driver *drv)
```

```
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct platform_driver *pdrv = to_platform_driver(drv);
5 
6     /*When driver_override is set,only bind to the matching driver*/
7     if (pdev->driver_override)
8     return !strcmp(pdev->driver_override, drv->name);
9 
10    /* Attempt an OF style match first */
11    if (of_driver_match_device(dev, drv))
12        return 1;
13 
14    /* Then try ACPI style match */
15    if (acpi_driver_match_device(dev, drv))
16        return 1;
17 
18    /* Then try to match against the id table */
19    if (pdrv->id_table)
20        return platform_match_id(pdrv->id_table, pdev) != NULL;
21 
22    /* fall-back to driver name match */
23    return (strcmp(pdev->name, drv->name) == 0);
24 }
```

驱动和设备的匹配有四种方法，我们依次来看一下：

第 11~12 行，第一种匹配方式， OF 类型的匹配，也就是设备树采用的匹配方式，of_driver_match_device 函数定义在文件 include/linux/of_device.h 中。device_driver 结构体(表示设备驱动)中有个名为of_match_table的成员变量，此成员变量保存着驱动的compatible匹配表，设备树中的每个设备节点的 compatible 属性会和 of_match_table 表中的所有成员比较，查看是否有相同的条目，如果有的话就表示设备和此驱动匹配，设备和驱动匹配成功以后 probe 函数就会执行。

第 15~16 行，第二种匹配方式，ACPI 匹配方式。

第 19~20 行，第三种匹配方式，id_table 匹配，每个 platform_driver 结构体有一个 id_table 成员变量，顾名思义，保存了很多 id 信息。这些 id 信息存放着这个 platformd 驱动所只是的驱动类型。

第 23 行，第四种匹配方式，如果第三种匹配方式的 id_table 不存在的话就直接比较驱动和设备的 name 字段，看看是不是相等，如果相等的话就匹配成功。

对于支持设备树的 Linux 版本号，一般设备驱动为了兼容性都支持设备树和无设备树两种匹配方式。也就是第一种匹配方式一般都会存在，第三种和第四种只要存在一种就可以，一般用的最多的还是第四种，也就是直接比较驱动和设备的 name 字段，毕竟这种方式最简单了。

54.2.2 platform 驱动

platform_driver 结构体表示 platform 驱动，此结构体定义在文件 include/linux/platform_device.h 中，内容如下：

示例代码 54.2.2.1 platform_driver 结构体

```
1  struct platform_driver {
2      int (*probe)(struct platform_device *);
3      int (*remove)(struct platform_device *);
4      void (*shutdown)(struct platform_device *);
5      int (*suspend)(struct platform_device *, pm_message_t state);
6      int (*resume)(struct platform_device *);
7      struct device_driver driver;
8      const struct platform_device_id *id_table;
9      bool prevent_deferred_probe;
10 };
```

第 2 行，probe 函数，当驱动与设备匹配成功以后 probe 函数就会执行，非常重要的函数！！一般驱动的提供者会编写，如果自己要编写一个全新的驱动，那么 probe 就需要自行事项。

第 7 行，driver 成员，为 device_driver 结构体变量，Linux 内核里面大量使用到了面向对象的思维，device_driver 相当于基类，提供了最基础的驱动框架。plaform_driver 继承了这个基类，然后在此基础上又添加了一些特有的成员变量。

第 8 行，id_table 表，也就是我们上一小节讲解 platform 总线匹配驱动和设备的时候采用的第三种方法，id_table 是个表(也就是数组)，每个元素的类型为 platform_device_id，platform_device_id 结构体内容如下：

示例代码 54.2.2.2 platform_device_id 结构体

```
1 struct platform_device_id {
2     char name[PLATFORM_NAME_SIZE];
3     kernel_ulong_t driver_data;
4 };
```

device_driver 结构体定义在 include/linux/device.h，device_driver 结构体内容如下：

示例代码 54.2.2.3 device_driver 结构体

```
1  struct device_driver {
2   const char          *name;
3   struct bus_type     *bus;
4
5   struct module       *owner;
6   const char          *mod_name;  /* used for built-in modules */
7
8   bool suppress_bind_attrs;   /* disables bind/unbind via sysfs */
9
10  const struct of_device_id       *of_match_table;
11  const struct acpi_device_id     *acpi_match_table;
12
13  int (*probe) (struct device *dev);
14  int (*remove) (struct device *dev);
15  void (*shutdown) (struct device *dev);
16  int (*suspend) (struct device *dev, pm_message_t state);
17  int (*resume) (struct device *dev);
```

```
18  const struct attribute_group **groups;
19
20  const struct dev_pm_ops *pm;
21
22  struct driver_private *p;
23 };
```

第 10 行，of_match_table 就是采用设备树的时候驱动使用的匹配表，同样是数组，每个匹配项都为 of_device_id 结构体类型，此结构体定义在文件 include/linux/mod_devicetable.h 中，内容如下：

示例代码 54.2.2.4 of_device_id 结构体

```
1 struct of_device_id {
2   char            name[32];
3   char            type[32];
4   char            compatible[128];
5   const void      *data;
6 };
```

第 4 行的 compatible 非常重要，因为对于设备树而言，就是通过设备节点的 compatible 属性值和 of_match_table 中每个项目的 compatible 成员变量进行比较，如果有相等的就表示设备和此驱动匹配成功。

在编写 platform 驱动的时候，首先定义一个 platform_driver 结构体变量，然后实现结构体中的各个成员变量，重点是实现匹配方法以及 probe 函数。当驱动和设备匹配成功以后 probe 函数就会执行，具体的驱动程序在 probe 函数里面编写，比如字符设备驱动等等。

当我们定义并初始化好 platform_driver 结构体变量以后，需要在驱动入口函数里面调用 platform_driver_register 函数向 Linux 内核注册一个 platform 驱动，platform_driver_register 函数原型如下所示：

```
int platform_driver_register (struct platform_driver     *driver)
```

函数参数和返回值含义如下：

driver：要注册的 platform 驱动。

返回值：负数，失败；0，成功。

还需要在驱动卸载函数中通过 platform_driver_unregister 函数卸载 platform 驱动，platform_driver_unregister 函数原型如下：

```
void platform_driver_unregister(struct platform_driver *drv)
```

函数参数和返回值含义如下：

drv：要卸载的 platform 驱动。

返回值：无。

platform 驱动框架如下所示：

示例代码 54.2.2.5 platform 驱动框架

```
   /* 设备结构体 */
1  struct xxx_dev{
2      struct cdev cdev;
3      /* 设备结构体其他具体内容 */
4  };
5
```

```
6  struct xxx_dev xxxdev;   /* 定义个设备结构体变量 */
7
8  static int xxx_open(struct inode *inode, struct file *filp)
9  {
10     /* 函数具体内容 */
11     return 0;
12 }
13
14 static ssize_t xxx_write(struct file *filp, const char __user *buf,
                             size_t cnt, loff_t *offt)
15 {
16     /* 函数具体内容 */
17     return 0;
18 }
19
20 /*
21  * 字符设备驱动操作集
22  */
23 static struct file_operations xxx_fops = {
24     .owner = THIS_MODULE,
25     .open = xxx_open,
26     .write = xxx_write,
27 };
28
29 /*
30  * platform 驱动的 probe 函数
31  * 驱动与设备匹配成功以后此函数就会执行
32  */
33 static int xxx_probe(struct platform_device *dev)
34 {
35     ......
36     cdev_init(&xxxdev.cdev, &xxx_fops); /* 注册字符设备驱动 */
37     /* 函数具体内容 */
38     return 0;
39 }
40
41 static int xxx_remove(struct platform_device *dev)
42 {
43     ......
44     cdev_del(&xxxdev.cdev);/*  删除 cdev */
45     /* 函数具体内容 */
46     return 0;
47 }
```

```
48
49 /* 匹配列表 */
50 static const struct of_device_id xxx_of_match[] = {
51     { .compatible = "xxx-gpio" },
52     { /* Sentinel */ }
53 };
54
55 /*
56  * platform 平台驱动结构体
57  */
58 static struct platform_driver xxx_driver = {
59     .driver = {
60         .name       = "xxx",
61         .of_match_table = xxx_of_match,
62     },
63     .probe      = xxx_probe,
64     .remove     = xxx_remove,
65 };
66
67 /* 驱动模块加载 */
68 static int __init xxxdriver_init(void)
69 {
70     return platform_driver_register(&xxx_driver);
71 }
72
73 /* 驱动模块卸载 */
74 static void __exit xxxdriver_exit(void)
75 {
76         platform_driver_unregister(&xxx_driver);
77 }
78
79 module_init(xxxdriver_init);
80 module_exit(xxxdriver_exit);
81 MODULE_LICENSE("GPL");
82 MODULE_AUTHOR("zuozhongkai");
```

第 1~27 行，传统的字符设备驱动，所谓的 platform 驱动并不是独立于字符设备驱动、块设备驱动和网络设备驱动之外的其他种类的驱动。platform 只是为了驱动的分离与分层而提出来的一种框架，其驱动的具体实现还是需要字符设备驱动、块设备驱动或网络设备驱动。

第 33~39 行，xxx_probe 函数，当驱动和设备匹配成功以后此函数就会执行，以前在驱动入口 init 函数里面编写的字符设备驱动程序就全部放到此 probe 函数里面。比如注册字符设备驱动、添加 cdev、创建类等等。

第 41~47 行，xxx_remove 函数，platform_driver 结构体中的 remove 成员变量，当关闭 platfor备驱动的时候此函数就会执行，以前在驱动卸载 exit 函数里面要做的事情就放到此函数中来。比如，使用 iounmap 释放内存、删除 cdev，注销设备号等等。

第 50~53 行，xxx_of_match 匹配表，如果使用设备树的话将通过此匹配表进行驱动和设备的匹配。第 51 行设置了一个匹配项，此匹配项的 compatible 值为“xxx-gpio”，因此当设备树中设备节点的 compatible 属性值为“xxx-gpio”的时候此设备就会与此驱动匹配。第 52 行是一个标记，of_device_id 表最后一个匹配项必须是空的。

第 58~65 行，定义一个 platform_driver 结构体变量 xxx_driver，表示 platform 驱动，第 59~62 行设置 paltform_driver 中的 device_driver 成员变量的 name 和 of_match_table 这两个属性。其中 name 属性用于传统的驱动与设备匹配，也就是检查驱动和设备的 name 字段是不是相同。of_match_table 属性就是用于设备树下的驱动与设备检查。对于一个完整的驱动程序，必须提供有设备树和无设备树两种匹配方法。最后 63 和 64 这两行设置 probe 和 remove 这两成员变量。

第 68~71 行，驱动入口函数，调用 platform_driver_register 函数向 Linux 内核注册一个 platform 驱动，也就是上面定义的 xxx_driver 结构体变量。

第 74~77 行，驱动出口函数，调用 platform_driver_unregister 函数卸载前面注册的 platform 驱动。

总体来说，platform 驱动还是传统的字符设备驱动、块设备驱动或网络设备驱动，只是套上了一张“platform”这张皮皮，目的是为了使用总线、驱动和设备这个驱动模型来实现驱动的分离与分层。

54.2.3 platform 设备

platform 驱动已经准备好了，我们还需要 platform 设备，否则的话单单一个驱动也做不了什么。platform_device 这个结构体表示 platform 设备，这里我们要注意，如果内核支持设备树的话就不要在使用 platform_device 来描述设备了，因为改用设备树去描述了。当然了，你如果一定要用 platform_device 来描述设备信息的话也是可以的。platform_device 结构体定义在文件 include/linux/platform_device.h 中，结构体内容如下：

示例代码 54.2.3.1 platform_device 结构体代码段

```
22 struct platform_device {
23     const char  *name;
24     int     id;
25     bool        id_auto;
26     struct device   dev;
27     u32     num_resources;
28     struct resource *resource;
29 
30     const struct platform_device_id *id_entry;
31     char *driver_override; /* Driver name to force a match */
32 
33     /* MFD cell pointer */
34     struct mfd_cell *mfd_cell;
35 
36     /* arch specific additions */
37     struct pdev_archdata    archdata;
```

```
38 };
```

第 23 行，name 表示设备名字，要和所使用的 platform 驱动的 name 字段相同，否则的话设备就无法匹配到对应的驱动。比如对应的 platform 驱动的 name 字段为“xxx-gpio”，那么此 name 字段也要设置为“xxx-gpio”。

第 27 行，num_resources 表示资源数量，一般为第 28 行 resource 资源的大小。

第 28 行，resource 表示资源，也就是设备信息，比如外设寄存器等。Linux 内核使用 resource 结构体表示资源，resource 结构体内容如下：

示例代码 54.2.3.2 resource 结构体代码段

```
18 struct resource {
19     resource_size_t   start;
20     resource_size_t   end;
21     const char        *name;
22     unsigned long     flags;
23     struct resource   *parent, *sibling, *child;
24 };
```

start 和 end 分别表示资源的起始和终止信息，对于内存类的资源，就表示内存起始和终止地址，name 表示资源名字，flags 表示资源类型，可选的资源类型都定义在了文件 include/linux/ioport.h 里面，如下所示：

示例代码 54.2.3.3 资源类型

```
29  #define IORESOURCE_BITS          0x000000ff  /* Bus-specific bits */
30
31  #define IORESOURCE_TYPE_BITS     0x00001f00  /* Resource type    */
32  #define IORESOURCE_IO            0x00000100  /* PCI/ISA I/O ports */
33  #define IORESOURCE_MEM           0x00000200
34  #define IORESOURCE_REG           0x00000300  /* Register offsets */
35  #define IORESOURCE_IRQ           0x00000400
36  #define IORESOURCE_DMA           0x00000800
37  #define IORESOURCE_BUS           0x00001000
......
104 /* PCI control bits.  Shares IORESOURCE_BITS with above PCI ROM.  */
105 #define IORESOURCE_PCI_FIXED     (1<<4)  /* Do not move resource */
```

在以前不支持设备树的Linux版本中，用户需要编写platform_device变量来描述设备信息，然后使用 platform_device_register 函数将设备信息注册到 Linux 内核中，此函数原型如下所示：

```
int platform_device_register(struct platform_device *pdev)
```

函数参数和返回值含义如下：

pdev：要注册的 platform 设备。

返回值：负数，失败；0，成功。

如果不再使用 platform 的话可以通过 platform_device_unregister 函数注销掉相应的 platform 设备，platform_device_unregister 函数原型如下：

```
void platform_device_unregister(struct platform_device *pdev)
```

函数参数和返回值含义如下：

pdev：要注销的 platform 设备。

返回值：无。

platform 设备信息框架如下所示：

示例代码 54.2.3.4 platform 设备框架

```
1  /* 寄存器地址定义*/
2  #define PERIPH1_REGISTER_BASE    (0X20000000) /* 外设 1 寄存器首地址 */
3  #define PERIPH2_REGISTER_BASE    (0X020E0068) /* 外设 2 寄存器首地址 */
4  #define REGISTER_LENGTH          4
5  
6  /* 资源 */
7  static struct resource xxx_resources[] = {
8      [0] = {
9          .start  = PERIPH1_REGISTER_BASE,
10         .end    = (PERIPH1_REGISTER_BASE + REGISTER_LENGTH - 1),
11         .flags  = IORESOURCE_MEM,
12     },
13     [1] = {
14         .start  = PERIPH2_REGISTER_BASE,
15         .end    = (PERIPH2_REGISTER_BASE + REGISTER_LENGTH - 1),
16         .flags  = IORESOURCE_MEM,
17     },
18 };
19 
20 /* platform 设备结构体 */
21 static struct platform_device xxxdevice = {
22     .name = "xxx-gpio",
23     .id = -1,
24     .num_resources = ARRAY_SIZE(xxx_resources),
25     .resource = xxx_resources,
26 };
27 
28 /* 设备模块加载 */
29 static int __init xxxdevice_init(void)
30 {
31     return platform_device_register(&xxxdevice);
32 }
33 
34 /* 设备模块注销 */
35 static void __exit xxx_resourcesdevice_exit(void)
36 {
37     platform_device_unregister(&xxxdevice);
38 }
39 
```

```
40 module_init(xxxdevice_init);
41 module_exit(xxxdevice_exit);
42 MODULE_LICENSE("GPL");
43 MODULE_AUTHOR("zuozhongkai");
```

第 7~18 行，数组 xxx_resources 表示设备资源，一共有两个资源，分别为设备外设 1 和外设 2 的寄存器信息。因此 flags 都为 IORESOURCE_MEM，表示资源为内存类型的。

第 21~26 行，platform 设备结构体变量，注意 name 字段要和所使用的驱动中的 name 字段一致，否则驱动和设备无法匹配成功。num_resources 表示资源大小，其实就是数组 xxx_resources 的元素数量，这里用 ARRAY_SIZE 来测量一个数组的元素个数。

第 29~32 行，设备模块加载函数，在此函数中调用 platform_device_register 向 Linux 内核注册 platform 设备。

第 35~38 行，设备模块卸载函数，在此函数中调用 platform_device_unregister 从 Linux 内核中卸载 platform 设备。

示例代码 54.2.3.4 主要是在不支持设备树的 Linux 版本中使用的，当 Linux 内核支持了设备树以后就不需要用户手动去注册 platform 设备了。因为设备信息都放到了设备树中去描述，Linux 内核启动的时候会从设备树中读取设备信息，然后将其组织成 platform_device 形式，至于设备树到 platform_device 的具体过程就不去详细的追究了，感兴趣的可以去看一下，网上也有很多博客详细的讲解了整个过程。

关于 platform 下的总线、驱动和设备就讲解到这里，我们接下来就使用 platform 驱动框架来编写一个 LED 灯驱动，本章我们不使用设备树来描述设备信息，我们采用自定义 platform_device 这种"古老"方式来编写 LED 的设备信息。下一章我们来编写设备树下的 platform 驱动，这样我们就掌握了无设备树和有设备树这两种 platform 驱动的开发方式。

54.3 硬件原理图分析

本章实验我们只使用到 IMX6U-ALPHA 开发板上的 LED 灯，因此实验硬件原理图参考 8.3 小节即可。

54.4 试验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 17_platform。

本章实验我们需要编写一个驱动模块和一个设备模块，其中驱动模块是 platform 驱动程序，设备模块是 platform 的设备信息。当这两个模块都加载成功以后就会匹配成功，然后 platform 驱动模块中的 probe 函数就会执行，probe 函数中就是传统的字符设备驱动那一套。

54.4.1 platform 设备与驱动程序编写

新建名为"17_platform"的文件夹，然后在 17_platform 文件夹里面创建 vscode 工程，工作区命名为"platform"。新建名为 leddevice.c 和 leddriver.c 这两个文件，这两个文件分别为 LED 灯的 platform 设备文件和 LED 灯的 platform 的驱动文件。在 leddevice.c 中输入如下所示内容：

示例代码 54.4.1.1 leddevice.c 文件代码段

```
1 #include <linux/types.h>
2 #include <linux/kernel.h>
3 #include <linux/delay.h>
4 #include <linux/ide.h>
```

```
5   #include <linux/init.h>
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of_gpio.h>
12  #include <linux/semaphore.h>
13  #include <linux/timer.h>
14  #include <linux/irq.h>
15  #include <linux/wait.h>
16  #include <linux/poll.h>
17  #include <linux/fs.h>
18  #include <linux/fcntl.h>
19  #include <linux/platform_device.h>
20  #include <asm/mach/map.h>
21  #include <asm/uaccess.h>
22  #include <asm/io.h>
23  /***************************************************************
24  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
25  文件名        : leddevice.c
26  作者          : 左忠凯
27  版本          : V1.0
28  描述          : platform 设备
29  其他          : 无
30  论坛          : www.openedv.com
31  日志          : 初版 V1.0 2019/8/13 左忠凯创建
32  ***************************************************************/
33
34  /*
35   * 寄存器地址定义
36   */
37  #define CCM_CCGR1_BASE              (0X020C406C)
38  #define SW_MUX_GPIO1_IO03_BASE      (0X020E0068)
39  #define SW_PAD_GPIO1_IO03_BASE      (0X020E02F4)
40  #define GPIO1_DR_BASE               (0X0209C000)
41  #define GPIO1_GDIR_BASE             (0X0209C004)
42  #define REGISTER_LENGTH             4
43
44  /* @description   : 释放 flatform 设备模块的时候此函数会执行
45   * @param - dev   : 要释放的设备
46   * @return        : 无
47   */
```

```
48 static void led_release(struct device *dev)
49 {
50     printk("led device released!\r\n");
51 }
52 
53 /*
54  * 设备资源信息，也就是 LED0 所使用的所有寄存器
55  */
56 static struct resource led_resources[] = {
57     [0] = {
58         .start  = CCM_CCGR1_BASE,
59         .end    = (CCM_CCGR1_BASE + REGISTER_LENGTH - 1),
60         .flags  = IORESOURCE_MEM,
61     },
62     [1] = {
63         .start  = SW_MUX_GPIO1_IO03_BASE,
64         .end    = (SW_MUX_GPIO1_IO03_BASE + REGISTER_LENGTH - 1),
65         .flags  = IORESOURCE_MEM,
66     },
67     [2] = {
68         .start  = SW_PAD_GPIO1_IO03_BASE,
69         .end    = (SW_PAD_GPIO1_IO03_BASE + REGISTER_LENGTH - 1),
70         .flags  = IORESOURCE_MEM,
71     },
72     [3] = {
73         .start  = GPIO1_DR_BASE,
74         .end    = (GPIO1_DR_BASE + REGISTER_LENGTH - 1),
75         .flags  = IORESOURCE_MEM,
76     },
77     [4] = {
78         .start  = GPIO1_GDIR_BASE,
79         .end    = (GPIO1_GDIR_BASE + REGISTER_LENGTH - 1),
80         .flags  = IORESOURCE_MEM,
81     },
82 };
83 
84 
85 /*
86  * platform 设备结构体
87  */
88 static struct platform_device leddevice = {
89     .name = "imx6ul-led",
90     .id = -1,
```

```
91      .dev = {
92          .release = &led_release,
93      },
94      .num_resources = ARRAY_SIZE(led_resources),
95      .resource = led_resources,
96  };
97
98  /*
99   * @description   : 设备模块加载
100  * @param         : 无
101  * @return        : 无
102  */
103 static int __init leddevice_init(void)
104 {
105     return platform_device_register(&leddevice);
106 }
107
108 /*
109  * @description   : 设备模块注销
110  * @param         : 无
111  * @return        : 无
112  */
113 static void __exit leddevice_exit(void)
114 {
115     platform_device_unregister(&leddevice);
116 }
117
118 module_init(leddevice_init);
119 module_exit(leddevice_exit);
120 MODULE_LICENSE("GPL");
121 MODULE_AUTHOR("zuozhongkai");
```

leddevice.c 文件内容就是按照示例代码 54.2.3.4 的 platform 设备模板编写的。

第 56~82 行，led_resources 数组，也就是设备资源，描述了 LED 所要使用到的寄存器信息，也就是 IORESOURCE_MEM 资源。

第 88~96，platform 设备结构体变量 leddevice，这里要注意 name 字段为“imx6ul-led”，所以稍后编写 platform 驱动中的 name 字段也要为“imx6ul-led”，否则设备和驱动匹配失败。

第 103~106 行，设备模块加载函数，在此函数里面通过 platform_device_register 向 Linux 内核注册 leddevice 这个 platform 设备。

第 113~116 行，设备模块卸载函数，在此函数里面通过 platform_device_unregister 从 Linux 内核中删除掉 leddevice 这个 platform 设备。

leddevice.c 文件编写完成以后就编写 leddriver.c 这个 platform 驱动文件，在 leddriver.c 里面输入如下内容：

示例代码 54.4.1.2 leddriver.c 文件代码段

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of_gpio.h>
12  #include <linux/semaphore.h>
13  #include <linux/timer.h>
14  #include <linux/irq.h>
15  #include <linux/wait.h>
16  #include <linux/poll.h>
17  #include <linux/fs.h>
18  #include <linux/fcntl.h>
19  #include <linux/platform_device.h>
20  #include <asm/mach/map.h>
21  #include <asm/uaccess.h>
22  #include <asm/io.h>
23  /***************************************************************
24  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
25  文件名      : leddriver.c
26  作者        : 左忠凯
27  版本        : V1.0
28  描述        : platform驱动
29  其他        : 无
30  论坛        : www.openedv.com
31  日志        : 初版 V1.0 2019/8/13 左忠凯创建
32  ***************************************************************/
33  
34  #define LEDDEV_CNT      1           /* 设备号长度   */
35  #define LEDDEV_NAME     "platled"   /* 设备名字     */
36  #define LEDOFF          0
37  #define LEDON           1
38  
39  /* 寄存器名 */
40  static void __iomem *IMX6U_CCM_CCGR1;
41  static void __iomem *SW_MUX_GPIO1_IO03;
42  static void __iomem *SW_PAD_GPIO1_IO03;
43  static void __iomem *GPIO1_DR;
```

```
44  static void __iomem *GPIO1_GDIR;
45
46  /* leddev 设备结构体 */
47  struct leddev_dev{
48      dev_t devid;            /* 设备号     */
49      struct cdev cdev;       /* cdev       */
50      struct class *class;    /* 类         */
51      struct device *device;  /* 设备       */
52      int major;              /* 主设备号   */
53  };
54
55  struct leddev_dev leddev;   /* led 设备   */
56
57  /*
58   * @description   : LED 打开/关闭
59   * @param - sta   : LEDON(0) 打开 LED，LEDOFF(1) 关闭 LED
60   * @return        : 无
61   */
62  void led0_switch(u8 sta)
63  {
64      u32 val = 0;
65      if(sta == LEDON){
66          val = readl(GPIO1_DR);
67          val &= ~(1 << 3);
68          writel(val, GPIO1_DR);
69      }else if(sta == LEDOFF){
70          val = readl(GPIO1_DR);
71          val|= (1 << 3);
72          writel(val, GPIO1_DR);
73      }
74  }
75
76  /*
77   * @description   : 打开设备
78   * @param – inode : 传递给驱动的 inode
79   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
80   *                 一般在 open 的时候将 private_data 指向设备结构体。
81   * @return        : 0 成功;其他 失败
82   */
83  static int led_open(struct inode *inode, struct file *filp)
84  {
85      filp->private_data = &leddev; /* 设置私有数据  */
86      return 0;
```

```
87  }
88
89  /*
90   * @description   : 向设备写数据
91   * @param – filp  : 设备文件，表示打开的文件描述符
92   * @param - buf   : 要写给设备写入的数据
93   * @param - cnt   : 要写入的数据长度
94   * @param - offt  : 相对于文件首地址的偏移
95   * @return        : 写入的字节数，如果为负值，表示写入失败
96   */
97  static ssize_t led_write(struct file *filp, const char __user *buf, 
                             size_t cnt, loff_t *offt)
98  {
99      int retvalue;
100     unsigned char databuf[1];
101     unsigned char ledstat;
102
103     retvalue = copy_from_user(databuf, buf, cnt);
104     if(retvalue < 0) {
105         return -EFAULT;
106     }
107
108     ledstat = databuf[0];       /* 获取状态值   */
109     if(ledstat == LEDON) {
110         led0_switch(LEDON);     /* 打开 LED 灯   */
111     }else if(ledstat == LEDOFF) {
112         led0_switch(LEDOFF);    /* 关闭 LED 灯   */
113     }
114     return 0;
115 }
116
117 /* 设备操作函数 */
118 static struct file_operations led_fops = {
119     .owner = THIS_MODULE,
120     .open = led_open,
121     .write = led_write,
122 };
123
124 /*
125  * @description   : flatform 驱动的 probe 函数，当驱动与设备
126  *                 匹配以后此函数就会执行
127  * @param - dev   : platform 设备
128  * @return        : 0，成功;其他负值,失败
```

```
129  */
130 static int led_probe(struct platform_device *dev)
131 {
132     int i = 0;
133     int ressize[5];
134     u32 val = 0;
135     struct resource *ledsource[5];
136 
137     printk("led driver and device has matched!\r\n");
138     /* 1、获取资源 */
139     for (i = 0; i < 5; i++) {
140         ledsource[i] = platform_get_resource(dev, IORESOURCE_MEM, i);
141         if (!ledsource[i]) {
142             dev_err(&dev->dev, "No MEM resource for always on\n");
143             return -ENXIO;
144         }
145         ressize[i] = resource_size(ledsource[i]);
146     }
147 
148     /* 2、初始化 LED */
149     /* 寄存器地址映射 */
150     IMX6U_CCM_CCGR1 = ioremap(ledsource[0]->start, ressize[0]);
151     SW_MUX_GPIO1_IO03 = ioremap(ledsource[1]->start, ressize[1]);
152     SW_PAD_GPIO1_IO03 = ioremap(ledsource[2]->start, ressize[2]);
153     GPIO1_DR = ioremap(ledsource[3]->start, ressize[3]);
154     GPIO1_GDIR = ioremap(ledsource[4]->start, ressize[4]);
155 
156     val = readl(IMX6U_CCM_CCGR1);
157     val &= ~(3 << 26);              /* 清除以前的设置   */
158     val |= (3 << 26);               /* 设置新值         */
159     writel(val, IMX6U_CCM_CCGR1);
160 
161     /* 设置 GPIO1_IO03 复用功能，将其复用为 GPIO1_IO03 */
162     writel(5, SW_MUX_GPIO1_IO03);
163     writel(0x10B0, SW_PAD_GPIO1_IO03);
164 
165     /* 设置 GPIO1_IO03 为输出功能 */
166     val = readl(GPIO1_GDIR);
167     val &= ~(1 << 3);               /* 清除以前的设置   */
168     val |= (1 << 3);                /* 设置为输出       */
169     writel(val, GPIO1_GDIR);
170 
171     /* 默认关闭 LED1 */
```

```
172     val = readl(GPIO1_DR);
173     val |= (1 << 3) ;
174     writel(val, GPIO1_DR);
175 
176     /* 注册字符设备驱动 */
177     /*1、创建设备号 */
178     if (leddev.major) {       /*  定义了设备号    */
179         leddev.devid = MKDEV(leddev.major, 0);
180         register_chrdev_region(leddev.devid, LEDDEV_CNT,
                                    LEDDEV_NAME);
181     } else {                  /* 没有定义设备号    */
182         alloc_chrdev_region(&leddev.devid, 0, LEDDEV_CNT,
                                 LEDDEV_NAME);
183         leddev.major = MAJOR(leddev.devid);
184     }
185 
186     /* 2、初始化 cdev */
187     leddev.cdev.owner = THIS_MODULE;
188     cdev_init(&leddev.cdev, &led_fops);
189 
190     /* 3、添加一个 cdev */
191     cdev_add(&leddev.cdev, leddev.devid, LEDDEV_CNT);
192 
193     /* 4、创建类 */
194     leddev.class = class_create(THIS_MODULE, LEDDEV_NAME);
195     if (IS_ERR(leddev.class)) {
196         return PTR_ERR(leddev.class);
197     }
198 
199     /* 5、创建设备 */
200     leddev.device = device_create(leddev.class, NULL, leddev.devid,
                                       NULL, LEDDEV_NAME);
201     if (IS_ERR(leddev.device)) {
202         return PTR_ERR(leddev.device);
203     }
204 
205     return 0;
206 }
207 
208 /*
209  * @description   :移除 platform 驱动的时候此函数会执行
210  * @param - dev   : platform 设备
211  * @return        : 0，成功;其他负值,失败
```

```
212  */
213 static int led_remove(struct platform_device *dev)
214 {
215     iounmap(IMX6U_CCM_CCGR1);
216     iounmap(SW_MUX_GPIO1_IO03);
217     iounmap(SW_PAD_GPIO1_IO03);
218     iounmap(GPIO1_DR);
219     iounmap(GPIO1_GDIR);
220 
221     cdev_del(&leddev.cdev);     /*  删除 cdev */
222     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
223     device_destroy(leddev.class, leddev.devid);
224     class_destroy(leddev.class);
225     return 0;
226 }
227 
228 /* platform 驱动结构体 */
229 static struct platform_driver led_driver = {
230     .driver     = {
231         .name   = "imx6ul-led",         /* 驱动名字，用于和设备匹配 */
232     },
233     .probe      = led_probe,
234     .remove     = led_remove,
235 };
236 
237 /*
238  * @description   : 驱动模块加载函数
239  * @param         : 无
240  * @return        : 无
241  */
242 static int __init leddriver_init(void)
243 {
244     return platform_driver_register(&led_driver);
245 }
246 
247 /*
248  * @description   : 驱动模块卸载函数
249  * @param         : 无
250  * @return        : 无
251  */
252 static void __exit leddriver_exit(void)
253 {
254     platform_driver_unregister(&led_driver);
```

```
255 }
256
257 module_init(leddriver_init);
258 module_exit(leddriver_exit);
259 MODULE_LICENSE("GPL");
260 MODULE_AUTHOR("zuozhongkai");
```

leddriver.c 文件内容就是按照示例代码 54.2.2.5 的 platform 驱动模板编写的。

第 34~122 行，传统的字符设备驱动。

第 130~206 行，probe 函数，当设备和驱动匹配以后此函数就会执行，当匹配成功以后会在终端上输出“led driver and device has matched!”这样语句。在 probe 函数里面初始化 LED、注册字符设备驱动。也就是将原来在驱动加载函数里面做的工作全部放到 probe 函数里面完成。

第 213~226 行，remobe 函数，当卸载 platform 驱动的时候此函数就会执行。在此函数里面释放内存、注销字符设备等。也就是将原来驱动卸载函数里面的工作全部都放到 remove 函数中完成。

第 229~235 行，platform_driver 驱动结构体，注意 name 字段为"imx6ul-led"，和我们在 leddevice.c 文件里面设置的设备 name 字段一致。

第 242~245 行，驱动模块加载函数，在此函数里面通过 platform_driver_register 向 Linux 内核注册 led_driver 驱动。

第 252~255 行，驱动模块卸载函数，在此函数里面通过 platform_driver_unregister 从 Linux 内核卸载 led_driver 驱动。

54.4.2 测试 APP 编写

测试 APP 的内容很简单，就是打开和关闭 LED 灯，新建 ledApp.c 这个文件，然后在里面输入如下内容：

示例代码 54.4.2.1 ledApp.c 文件代码段

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名       : ledApp.c
11 作者         : 左忠凯
12 版本         : V1.0
13 描述         : platform 驱动驱测试 APP。
14 其他         : 无
15 使用方法     : ./ledApp /dev/platled  0 关闭 LED
16                ./ledApp /dev/platled  1 打开 LED
17 论坛         : www.openedv.com
18 日志         : 初版 V1.0 2019/8/16 左忠凯创建
```

```
19 ***************************************************************/
20 #define LEDOFF  0
21 #define LEDON   1
22
23 /*
24  * @description    : main 主程序
25  * @param - argc   : argv 数组元素个数
26  * @param - argv   : 具体参数
27  * @return         : 0 成功;其他 失败
28  */
29 int main(int argc, char *argv[])
30 {
31     int fd, retvalue;
32     char *filename;
33     unsigned char databuf[2];
34
35     if(argc != 3){
36         printf("Error Usage!\r\n");
37         return -1;
38     }
39
40     filename = argv[1];
41     /* 打开 led 驱动 */
42     fd = open(filename, O_RDWR);
43     if(fd < 0){
44         printf("file %s open failed!\r\n", argv[1]);
45         return -1;
46     }
47
48     databuf[0] = atoi(argv[2]); /* 要执行的操作：打开或关闭 */
49     retvalue = write(fd, databuf, sizeof(databuf));
50     if(retvalue < 0){
51         printf("LED Control Failed!\r\n");
52         close(fd);
53         return -1;
54     }
55
56     retvalue = close(fd); /* 关闭文件 */
57     if(retvalue < 0){
58         printf("file %s close failed!\r\n", argv[1]);
59         return -1;
60     }
61     return 0;
```

```
62 }
```

ledApp.c 文件内容很简单，就是控制 LED 灯的亮灭，和第四十一章的测试 APP 基本一致，这里就不重复讲解了。

54.5 运行测试

54.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为“leddevice.o leddriver.o”，Makefile 内容如下所示：

示例代码 54.5.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := leddevice.o leddriver.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“leddevice.o leddriver.o”。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“leddevice.ko leddriver.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

54.4.2 运行测试

将上一小节编译出来 leddevice.ko、leddriver.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 leddevice.ko 设备模块和 leddriver.ko 这个驱动模块。

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe leddevice.ko   //加载设备模块
modprobe leddriver.ko   //加载驱动模块
```

根文件系统中/sys/bus/platform/目录下保存着当前板子 platform 总线下的设备和驱动，其中 devices 子目录为 platform 设备，drivers 子目录为 plartofm 驱动。查看/sys/bus/platform/devices/目录，看看我们的设备是否存在，我们在 leddevice.c 中设置 leddevice(platform_device 类型)的 name 字段为“imx6ul-led”，也就是设备名字为 imx6ul-led，因此肯定在/sys/bus/platform/devices/目录下存在一个名字“imx6ul-led”的文件，否则说明我们的设备模块加载失败，结果如图 54.4.2.1 所示：

```
/lib/modules/4.1.15 # ls /sys/bus/platform/devices/
20cc000.snvs:snvs-powerkey          imx6q-cpufreq
20cc000.snvs:snvs-poweroff          imx6ul-led      设备
20cc000.snvs:snvs-rtc-lp            key
```

图 54.4.2.1 imx6ul-led 设备

同理，查看/sys/bus/platform/drivers/目录，看一下驱动是否存在，我们在 leddriver.c 中设置 led_driver (platform_driver 类型)的 name 字段为“imx6ul-led”，因此会在/sys/bus/platform/drivers/目录下存在名为“imx6ul-led”这个文件，结果如图 54.4.2.2 所示：

```
/lib/modules/4.1.15 # ls /sys/bus/platform/drivers
gpio-mxc                imx6sx-pinctrl   驱动     smc911x
gpio-rc-recv            imx6ul-led                smc91x
```

图 54.4.2.2 imx6ul-led 驱动

驱动模块和设备模块加载成功以后 platform 总线就会进行匹配，当驱动和设备匹配成功以后就会输出如图 54.4.2.3 所示一行语句：

```
/lib/modules/4.1.15 # modprobe leddevice.ko
/lib/modules/4.1.15 # modprobe leddriver.ko
led driver and device has matched!      驱动和设备匹配成功
/lib/modules/4.1.15 #
```

图 54.4.2.3 驱动和设备匹配成功

驱动和设备匹配成功以后就可以测试 LED 灯驱动了，输入如下命令打开 LED 灯：

```
./ledApp /dev/platled 1   //打开 LED 灯
```

在输入如下命令关闭 LED 灯：

```
./ledApp /dev/platled 0   //关闭 LED 灯
```

观察一下 LED 灯能否打开和关闭，如果可以的话就说明驱动工作正常，如果要卸载驱动的话输入如下命令即可：

```
rmmod leddevice.ko
rmmod leddriver.ko
```

第五十五章 设备树下的 platform 驱动编写

上一章我们详细的讲解了 Linux 下的驱动分离与分层，以及总线、设备和驱动这样的驱动框架。基于总线、设备和驱动这样的驱动框架，Linux 内核提出来 platform 这个虚拟总线，相应的也有 platform 设备和 platform 驱动。上一章我们讲解了传统的、未采用设备树的 platform 设备和驱动编写方法。最新的 Linux 内核已经支持了设备树，因此在设备树下如何编写 platform 驱动就显得尤为重要，本章我们就来学习一下如何在设备树下编写 platform 驱动。

55.1 设备树下的 platform 驱动简介

platform 驱动框架分为总线、设备和驱动，其中总线不需要我们这些驱动程序员去管理，这个是 Linux 内核提供的，我们在编写驱动的时候只要关注于设备和驱动的具体实现即可。在未设备树的 Linux 内核下，我们需要分别编写并注册 platform_device 和 platform_driver，分别代表设备和驱动。在使用设备树的时候，设备的描述被放到了设备树中，因此 platform_device 就不需要我们去编写了，我们只需要实现 platform_driver 即可。在编写基于设备树的 platform 驱动的时候我们需要注意一下几点：

1、在设备树中创建设备节点

毫无疑问，肯定要先在设备树中创建设备节点来描述设备信息，重点是要设置好 compatible 属性的值，因为 platform 总线需要通过设备节点的 compatible 属性值来匹配驱动！这点要切记。比如，我们可以编写如下所示的设备节点来描述我们本章实验要用到的 LED 这个设备：

示例代码 55.1.1 gpioled 设备节点

```
1 gpioled {
2       #address-cells = <1>;
3       #size-cells = <1>;
4       compatible = "atkalpha-gpioled";
5       pinctrl-names = "default";
6       pinctrl-0 = <&pinctrl_led>;
7       led-gpio = <&gpio1 3 GPIO_ACTIVE_LOW>;
8       status = "okay";
9 };
```

示例 55.1.1 中的 gpioled 节点其实就是 45.4.1.2 小节中创建的 gpioled 设备节点，我们可以直接拿过来用。注意第 4 行的 compatible 属性值为“atkalpha-gpioled”，因此，我们一会在编写 platform 驱动的时候一定要设置 of_match_table 也有此值。

2、编写 platform 驱动的时候要注意兼容属性

上一章已经详细的讲解过了，在使用设备树的时候 platform 驱动会通过 of_match_table 来保存兼容性值，也就是表明此驱动兼容哪些设备。所以，of_match_table 将会尤为重要，比如本例程的 platform 驱动中 platform_driver 就可以按照如下所示设置：

示例代码 55.1.2 of_match_table 匹配表的设置

```
1  static const struct of_device_id leds_of_match[] = {
2      { .compatible = "atkalpha-gpioled" },  /* 兼容属性 */
3      { /* Sentinel */ }
4  };
5
6  MODULE_DEVICE_TABLE(of, leds_of_match);
7
8  static struct platform_driver leds_platform_driver = {
9      .driver = {
10         .name       = "imx6ul-led",
11         .of_match_table = leds_of_match,
12     },
```

```
13      .probe          = leds_probe,
14      .remove         = leds_remove,
15  };
```

第 1~4 行，of_device_id 表，也就是驱动的兼容表，是一个数组，每个数组元素为 of_device_id 类型。每个数组元素都是一个兼容属性，表示兼容的设备，一个驱动可以跟多个设备匹配。这里我们仅仅匹配了一个设备，那就是 55.1.1 中创建的 gpioled 这个设备。第 2 行的 compatible 值为"atkalpha-gpioled"，驱动中的 compatible 属性和设备中的 compatible 属性相匹配，因此驱动中对应的 probe 函数就会执行。注意第 3 行是一个空元素，在编写 of_device_id 的时候最后一个元素一定要为空！

第 6 行，通过 MODULE_DEVICE_TABLE 声明一下 leds_of_match 这个设备匹配表。

第 11 行，设置 platform_driver 中的 of_match_table 匹配表为上面创建的 leds_of_match，至此我们就设置好了 platform 驱动的匹配表了。

3、编写 platform 驱动

基于设备树的 platform 驱动和上一章无设备树的 platform 驱动基本一样，都是当驱动和设备匹配成功以后就会执行 probe 函数。我们需要在 probe 函数里面执行字符设备驱动那一套，当注销驱动模块的时候 remove 函数就会执行，都是大同小异的。

55.2 硬件原理图分析

本章实验我们只使用到 IMX6U-ALPHA 开发板上的 LED 灯，因此实验硬件原理图参考 8.3 小节即可。

55.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 18_dtsplatform。

本章实验我们编写基于设备树的 platform 驱动，所以需要在设备树中添加设备节点，然后我们只需要编写 platform 驱动即可。

55.3.1 修改设备树文件

首先修改设备树文件，加上我们需要的设备信息，本章我们就使用到一个 LED 灯，因此可以直接使用 45.4.1 小节编写的 gpioled 子节点即可，不需要在重复添加。

55.3.2 platform 驱动程序编写

设备已经准备好了，接下来就要编写相应的 platform 驱动了，新建名为"18_dtsplatform"的文件夹，然后在 18_dtsplatform 文件夹里面创建 vscode 工程，工作区命名为"dtsplatform"。新建名为 leddriver.c 的驱动文件，在 leddriver.c 中输入如下所示内容：

示例代码 55.3.2.1 leddriver.c 文件代码段

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
```

```
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of_gpio.h>
12  #include <linux/semaphore.h>
13  #include <linux/timer.h>
14  #include <linux/irq.h>
15  #include <linux/wait.h>
16  #include <linux/poll.h>
17  #include <linux/fs.h>
18  #include <linux/fcntl.h>
19  #include <linux/platform_device.h>
20  #include <asm/mach/map.h>
21  #include <asm/uaccess.h>
22  #include <asm/io.h>
23  /***************************************************************
24  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
25  文件名      : leddriver.c
26  作者        : 左忠凯
27  版本        : V1.0
28  描述        : 设备树下的 platform 驱动
29  其他        : 无
30  论坛        : www.openedv.com
31  日志        : 初版 V1.0 2019/8/13 左忠凯创建
32  ***************************************************************/
33  #define LEDDEV_CNT      1               /* 设备号长度   */
34  #define LEDDEV_NAME     "dtsplatled"    /* 设备名字     */
35  #define LEDOFF          0
36  #define LEDON           1
37
38  /* leddev 设备结构体 */
39  struct leddev_dev{
40      dev_t           devid;          /* 设备号          */
41      struct cdev     cdev;           /* cdev            */
42      struct class    *class;         /* 类              */
43      struct device *device;          /* 设备            */
44      int             major;          /* 主设备号        */
45      struct device_node *node;       /* LED 设备节点    */
46      int             led0;           /* LED 灯 GPIO 标号 */
47  };
48
49  struct leddev_dev leddev;           /* led 设备        */
50
```

```
51 /*
52  * @description   : LED 打开/关闭
53  * @param - sta   : LEDON(0) 打开 LED，LEDOFF(1) 关闭 LED
54  * @return        : 无
55  */
56 void led0_switch(u8 sta)
57 {
58     if (sta == LEDON )
59         gpio_set_value(leddev.led0, 0);
60     else if (sta == LEDOFF)
61         gpio_set_value(leddev.led0, 1);
62 }
63
64 /*
65  * @description   : 打开设备
66  * @param – inode : 传递给驱动的 inode
67  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
68  *                 一般在 open 的时候将 private_data 指向设备结构体。
69  * @return        : 0 成功;其他 失败
70  */
71 static int led_open(struct inode *inode, struct file *filp)
72 {
73     filp->private_data = &leddev; /* 设置私有数据  */
74     return 0;
75 }
76
77 /*
78  * @description   : 向设备写数据
79  * @param - filp  : 设备文件，表示打开的文件描述符
80  * @param - buf   : 要写给设备写入的数据
81  * @param - cnt   : 要写入的数据长度
82  * @param – offt  : 相对于文件首地址的偏移
83  * @return        : 写入的字节数，如果为负值，表示写入失败
84  */
85 static ssize_t led_write(struct file *filp, const char __user *buf,
                           size_t cnt, loff_t *offt)
86 {
87     int retvalue;
88     unsigned char databuf[2];
89     unsigned char ledstat;
90
91     retvalue = copy_from_user(databuf, buf, cnt);
92     if(retvalue < 0) {
```

```
93
94         printk("kernel write failed!\r\n");
95         return -EFAULT;
96     }
97
98     ledstat = databuf[0];
99     if (ledstat == LEDON) {
100        led0_switch(LEDON);
101    } else if (ledstat == LEDOFF) {
102        led0_switch(LEDOFF);
103    }
104    return 0;
105 }
106
107 /* 设备操作函数 */
108 static struct file_operations led_fops = {
109     .owner = THIS_MODULE,
110     .open = led_open,
111     .write = led_write,
112 };
113
114 /*
115  * @description   : flatform 驱动的 probe 函数，当驱动与
116  *                  设备匹配以后此函数就会执行
117  * @param - dev   : platform 设备
118  * @return        : 0，成功;其他负值,失败
119  */
120 static int led_probe(struct platform_device *dev)
121 {
122     printk("led driver and device was matched!\r\n");
123     /* 1、设置设备号 */
124     if (leddev.major) {
125         leddev.devid = MKDEV(leddev.major, 0);
126         register_chrdev_region(leddev.devid, LEDDEV_CNT,
                                LEDDEV_NAME);
127     } else {
128         alloc_chrdev_region(&leddev.devid, 0, LEDDEV_CNT,
                               LEDDEV_NAME);
129         leddev.major = MAJOR(leddev.devid);
130     }
131
132     /* 2、注册设备      */
133     cdev_init(&leddev.cdev, &led_fops);
```

```
134     cdev_add(&leddev.cdev, leddev.devid, LEDDEV_CNT);
135 
136     /* 3、创建类      */
137     leddev.class = class_create(THIS_MODULE, LEDDEV_NAME);
138     if (IS_ERR(leddev.class)) {
139         return PTR_ERR(leddev.class);
140     }
141 
142     /* 4、创建设备 */
143     leddev.device = device_create(leddev.class, NULL, leddev.devid,
                                     NULL, LEDDEV_NAME);
144     if (IS_ERR(leddev.device)) {
145         return PTR_ERR(leddev.device);
146     }
147 
148     /* 5、初始化 IO */
149     leddev.node = of_find_node_by_path("/gpioled");
150     if (leddev.node == NULL){
151         printk("gpioled node nost find!\r\n");
152         return -EINVAL;
153     }
154 
155     leddev.led0 = of_get_named_gpio(leddev.node, "led-gpio", 0);
156     if (leddev.led0 < 0) {
157         printk("can't get led-gpio\r\n");
158         return -EINVAL;
159     }
160 
161     gpio_request(leddev.led0, "led0");
162     gpio_direction_output(leddev.led0, 1); /*设置为输出，默认高电平 */
163     return 0;
164 }
165 
166 /*
167  * @description   : remove 函数，移除 platform 驱动的时候此函数会执行
168  * @param - dev   : platform 设备
169  * @return        : 0，成功;其他负值,失败
170  */
171 static int led_remove(struct platform_device *dev)
172 {
173     gpio_set_value(leddev.led0, 1); /* 卸载驱动的时候关闭 LED    */
174 
175     cdev_del(&leddev.cdev);          /*  删除 cdev              */
```

```
176     unregister_chrdev_region(leddev.devid, LEDDEV_CNT);
177     device_destroy(leddev.class, leddev.devid);
178     class_destroy(leddev.class);
179     return 0;
180 }
181 
182 /* 匹配列表 */
183 static const struct of_device_id led_of_match[] = {
184     { .compatible = "atkalpha-gpioled" },
185     { /* Sentinel */ }
186 };
187 
188 /* platform 驱动结构体 */
189 static struct platform_driver led_driver = {
190     .driver     = {
191         .name   = "imx6ul-led",          /* 驱动名字，用于和设备匹配  */
192         .of_match_table = led_of_match, /* 设备树匹配表            */
193     },
194     .probe      = led_probe,
195     .remove     = led_remove,
196 };
197 
198 /*
199  * @description   : 驱动模块加载函数
200  * @param         : 无
201  * @return        : 无
202  */
203 static int __init leddriver_init(void)
204 {
205     return platform_driver_register(&led_driver);
206 }
207 
208 /*
209  * @description   : 驱动模块卸载函数
210  * @param         : 无
211  * @return        : 无
212  */
213 static void __exit leddriver_exit(void)
214 {
215     platform_driver_unregister(&led_driver);
216 }
217 
218 module_init(leddriver_init);
```

```
219 module_exit(leddriver_exit);
220 MODULE_LICENSE("GPL");
221 MODULE_AUTHOR("zuozhongkai");
```

第 33~112 行，传统的字符设备驱动，没什么要说的。

第 120~164 行，platform 驱动的 probe 函数，当设备树中的设备节点与驱动之间匹配成功以后此函数就会执行，原来在驱动加载函数里面做的工作现在全部放到 probe 函数里面完成。

第 171~180 行，remobe 函数，当卸载 platform 驱动的时候此函数就会执行。在此函数里面释放内存、注销字符设备等，也就是将原来驱动卸载函数里面的工作全部都放到 remove 函数中完成。

第 183~186 行，匹配表，描述了此驱动都和什么样的设备匹配，第 184 行添加了一条值为"atkalpha-gpioled"的 compatible 属性值，当设备树中某个设备节点的 compatible 属性值也为"atkalpha-gpioled"的时候就会与此驱动匹配。

第 189~196 行，platform_driver 驱动结构体，191 行设置这个 platform 驱动的名字为"imx6ul-led"，因此，当驱动加载成功以后就会在/sys/bus/platform/drivers/目录下存在一个名为"imx6u-led"的文件。 第 192 行设置 of_match_table 为上面的 led_of_match。

第 203~206 行，驱动模块加载函数，在此函数里面通过 platform_driver_register 向 Linux 内核注册 led_driver 驱动。

第 213~216 行，驱动模块卸载函数，在此函数里面通过 platform_driver_unregister 从 Linux 内核卸载 led_driver 驱动。

55.3.3 编写测试 APP

测试 APP 就直接使用上一章 54.4.2 小节编写的 ledApp.c 即可。

55.4 运行测试

55.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为"leddriver.o"，Makefile 内容如下所示：

示例代码 55.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := leddriver.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为"leddriver.o"。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为"leddriver.o"的驱动模块文件。

2、编译测试 APP

测试 APP 直接使用上一章的 ledApp 这个测试软件即可。

55.4.2 运行测试

将上一小节编译出来 leddriver.ko 拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 leddriver.ko 这个驱动模块。

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe leddriver.ko   //加载驱动模块
```

驱动模块加载完成以后到/sys/bus/platform/drivers/目录下查看驱动是否存在，我们在 leddriver.c 中设置 led_driver (platform_driver 类型)的 name 字段为“imx6ul-led”，因此会在/sys/bus/platform/drivers/目录下存在名为“imx6ul-led”这个文件，结果如图 55.4.2.1 所示：

```
/lib/modules/4.1.15 # ls /sys/bus/platform/drivers
gpio-mxc              imx6sx-pinctrl        smc911x
gpio-rc-recv          imx6ul-led   驱动     smc91x
```

图 55.4.2.1 imx6ul-led 驱动

同理，在/sys/bus/platform/devices/目录下也存在 led 的设备文件，也就是设备树中 gpioled 这个节点，如图 55.4.2.2 所示：

```
/lib/modules/4.1.15 # ls /sys/bus/platform/devices/
20ca000.usbphy                 ci_hdrc.1
20cc000.snvs                   gpioled   设备
```

图 55.4.2.2 gpioled 设备

驱动和模块都存在，当驱动和设备匹配成功以后就会输出如图 55.4.2.3 所示一行语句：

```
/lib/modules/4.1.15 # modprobe leddriver.ko
led driver and device was matched!   驱动和设备匹配成功
/lib/modules/4.1.15 #
```

图 55.4.2.3 驱动和设备匹配成功

驱动和设备匹配成功以后就可以测试 LED 灯驱动了，输入如下命令打开 LED 灯：

```
./ledApp /dev/dtsplatled 1    //打开 LED 灯
```

在输入如下命令关闭 LED 灯：

```
./ledApp /dev/dtsplatled 0    //关闭 LED 灯
```

观察一下 LED 灯能否打开和关闭，如果可以的话就说明驱动工作正常，如果要卸载驱动的话输入如下命令即可：

```
rmmod leddriver.ko
```

第五十六章 Linux 自带的 LED 灯驱动实验

前面我们都是自己编写 LED 灯驱动，其实像 LED 灯这样非常基础的设备驱动，Linux 内核已经集成了。Linux 内核的 LED 灯驱动采用 platform 框架，因此我们只需要按照要求在设备树文件中添加相应的 LED 节点即可，本章我们就来学习如何使用 Linux 内核自带的 LED 驱动来驱动 I.MX6U-ALPHA 开发板上的 LED0。

56.1 Linux 内核自带 LED 驱动使能

上一章节我们编写基于设备树的 platform LED 灯驱动，其实 Linux 内核已经自带了 LED 灯驱动，要使用 Linux 内核自带的 LED 灯驱动首先得先配置 Linux 内核，使能自带的 LED 灯驱动，输入如下命令打开 Linux 配置菜单：

```
make menuconfig
```

按照如下路径打开 LED 驱动配置项：

```
-> Device Drivers
    -> LED Support (NEW_LEDS [=y])
        ->LED Support for GPIO connected LEDs
```

按照上述路径，选择“LED Support for GPIO connected LEDs”，将其编译进 Linux 内核，也即是在此选项上按下“Y”键，使此选项前面变为“<*>”，如图 56.1.1 所示：

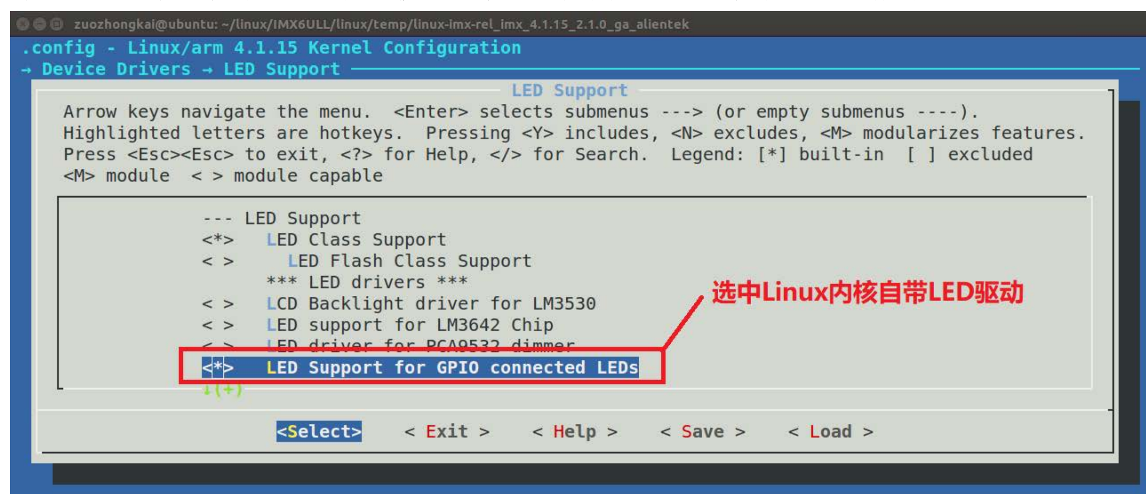


图 56.1.1 使能 LED 灯驱动

在“LED Support for GPIO connected LEDs”上按下可以打开此选项的帮助信息，如图 56.1.2 所示：

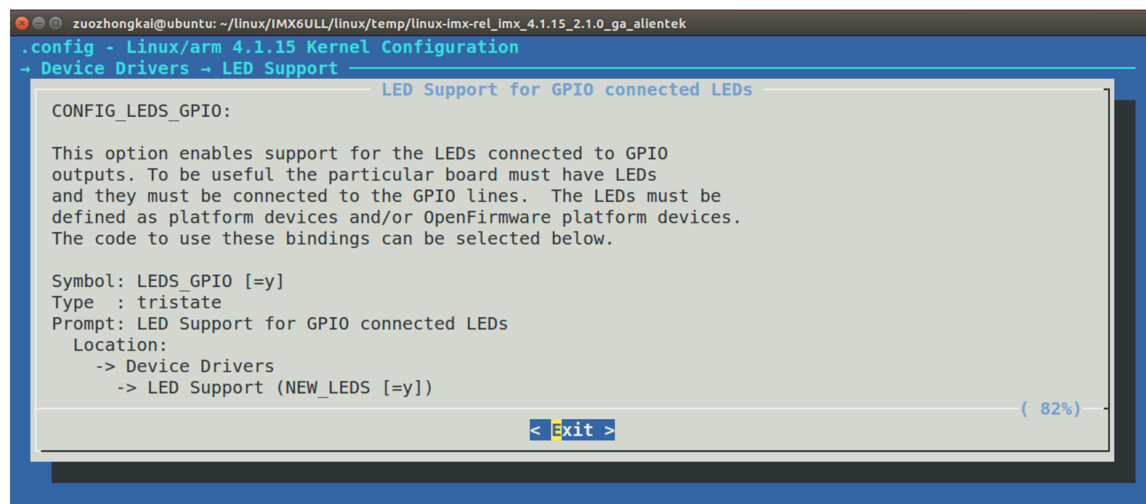


图 56.1.2 内部 LED 灯驱动帮助信息

从图 56.1.2 可以看出，把 Linux 内部自带的 LED 灯驱动编译进内核以后，CONFIG_LEDS_GPIO 就会等于‘y’，Linux 会根据 CONFIG_LEDS_GPIO 的值来选择如何编译 LED 灯驱动，如果为‘y’就将其编译进 Linux 内核。

配置好 Linux 内核以后退出配置界面，打开.config 文件，会找到“CONFIG_LEDS_GPIO=y”这一行，如图 56.1.3 所示：

```
zuozhongkai@ubuntu: ~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek
2725 #
2726 # LED drivers
2727 #
2728 # CONFIG_LEDS_LM3530 is not set
2729 # CONFIG_LEDS_LM3642 is not set
2730 # CONFIG_LEDS_PCA9532 is not set
2731 CONFIG_LEDS_GPIO=y
2732 # CONFIG_LEDS_LP3944 is not set
2733 # CONFIG_LEDS_LP5521 is not set
2734 # CONFIG_LEDS_LP5523 is not set
2735 # CONFIG_LEDS_LP5562 is not set
2736 # CONFIG_LEDS_LP8501 is not set
2737 # CONFIG_LEDS_LP8860 is not set
```

图 56.1.3 .config 文件内容

重新编译 Linux 内核，然后使用新编译出来的 zImage 镜像启动开发板。

56.2 Linux 内核自带 LED 驱动简介

56.2.1 LED 灯驱动框架分析

LED 灯驱动文件为/drivers/leds/leds-gpio.c，大家可以打开/drivers/leds/Makefile 这个文件，找到如下所示内容：

示例代码 56.2.1.1 /drivers/leds/Makefile 文件代码段

```
2  # LED Core
3  obj-$(CONFIG_NEW_LEDS)              += led-core.o
.....
23 obj-$(CONFIG_LEDS_GPIO_REGISTER)    += leds-gpio-register.o
24 obj-$(CONFIG_LEDS_GPIO)             += leds-gpio.o
25 obj-$(CONFIG_LEDS_LP3944)           += leds-lp3944.o
......
```

第 25 行，如果定义了 CONFIG_LEDS_GPIO 的话就会编译 leds-gpio.c 这个文件，在上一小节我们选择将 LED 驱动编译进 Linux 内核，在.config 文件中就会有“CONFIG_LEDS_GPIO=y”这一行，因此 leds-gpio.c 驱动文件就会被编译。

接下来我们看一下 leds-gpio.c 这个驱动文件，找到如下所示内容：

示例代码 56.2.1.2 leds-gpio.c 文件代码段

```
236 static const struct of_device_id of_gpio_leds_match[] = {
237     { .compatible = "gpio-leds", },
238     {},
239 };
......
290 static struct platform_driver gpio_led_driver = {
```

```
291     .probe      = gpio_led_probe,
292     .remove     = gpio_led_remove,
293     .driver     = {
294         .name   = "leds-gpio",
295         .of_match_table = of_gpio_leds_match,
296     },
297 };
298 
299 module_platform_driver(gpio_led_driver);
```

第 236~239 行，LED 驱动的匹配表，此表只有一个匹配项，compatible 内容为“gpio-leds”，因此设备树中的 LED 灯设备节点的 compatible 属性值也要为“gpio-leds”，否则设备和驱动匹配不成功，驱动就没法工作。

第 290~296 行，platform_driver 驱动结构体变量，可以看出，Linux 内核自带的 LED 驱动采用了 platform 框架。第 291 行可以看出 probe 函数为 gpio_led_probe，因此当驱动和设备匹配成功以后 gpio_led_probe 函数就会执行。从 294 行可以看出，驱动名字为“leds-gpio”，因此会在/sys/bus/platform/drivers 目录下存在一个名为“leds-gpio”的文件，如图 56.2.1.1 所示：

```
/lib/modules/4.1.15 # ls /sys/bus/platform/drivers
imx-gpcv2                  ldo2p5-dummy          sram
imx-i2c                    leds-gpio             stmpe-ts
```
自带LED驱动

图 56.2.1.1 leds-gpio 驱动文件

第 299 行通过 module_platform_driver 函数向 Linux 内核注册 gpio_led_driver 这个 platform 驱动。

56.2.2 module_platform_driver 函数简析

在上一小节中我们知道 LED 驱动会采用 module_platform_driver 函数向 Linux 内核注册 platform 驱动，其实在 Linux 内核中会大量采用 module_platform_driver 来完成向 Linux 内核注册 platform 驱动的操作。module_platform_driver 定义在 include/linux/platform_device.h 文件中，为一个宏，定义如下：

示例代码 56.2.2.1 module_platform_driver 函数

```
221 #define module_platform_driver(__platform_driver) \
222     module_driver(__platform_driver, platform_driver_register, \
223             platform_driver_unregister)
```

可以看出，module_platform_driver 依赖 module_driver，module_driver 也是一个宏，定义在 include/linux/device.h 文件中，内容如下：

示例代码 56.2.2.2 module_driver 函数

```
1260 #define module_driver(__driver, __register, __unregister, ...) \
1261 static int __init __driver##_init(void) \
1262 { \
1263    return __register(&(__driver) , ##__VA_ARGS__); \
1264 } \
1265 module_init(__driver##_init); \
1266 static void __exit __driver##_exit(void) \
1267 { \
```

```
1268    __unregister(&(__driver) , ##__VA_ARGS__); \
1269 } \
1270 module_exit(__driver##_exit);
```

借助示例代码 56.2.2.1 和示例代码 56.2.2.2，将：

```
module_platform_driver(gpio_led_driver)
```

展开以后就是：

```
static int __init gpio_led_driver_init(void)
{
    return platform_driver_register (&(gpio_led_driver));
}
module_init(gpio_led_driver_init);

static void __exit gpio_led_driver_exit(void)
{
    platform_driver_unregister (&(gpio_led_driver) );
}
module_exit(gpio_led_driver_exit);
```

上面的代码不就是标准的注册和删除 platform 驱动吗？因此 module_platform_driver 函数的功能就是完成 platform 驱动的注册和删除。

56.2.3 gpio_led_probe 函数简析

当驱动和设备匹配以后 gpio_led_probe 函数就会执行，此函数主要是从设备树中获取 LED 灯的 GPIO 信息，缩减后的函数内容如下所示：

示例代码 56.2.3.1 gpio_led_probe 函数

```
243 static int gpio_led_probe(struct platform_device *pdev)
244 {
245     struct gpio_led_platform_data *pdata = 
                                dev_get_platdata(&pdev->dev);
246     struct gpio_leds_priv *priv;
247     int i, ret = 0;
248 
249     if (pdata && pdata->num_leds) {     /* 非设备树方式     */
           /* 获取 platform_device 信息 */
           ......
268     } else {                           /* 采用设备树       */
269         priv = gpio_leds_create(pdev);
270         if (IS_ERR(priv))
271             return PTR_ERR(priv);
272     }
273 
274     platform_set_drvdata(pdev, priv);
275 
276     return 0;
```

```
277 }
```

第 269~271 行，如果使用设备树的话，使用 gpio_leds_create 函数从设备树中提取设备信息，获取到的 LED 灯 GPIO 信息保存在返回值中，gpio_leds_create 函数内容如下：

示例代码 56.2.3.2 gpio_leds_create 函数

```
167 static struct gpio_leds_priv *gpio_leds_create(struct
                                            platform_device *pdev)
168 {
169     struct device *dev = &pdev->dev;
170     struct fwnode_handle *child;
171     struct gpio_leds_priv *priv;
172     int count, ret;
173     struct device_node *np;
174 
175     count = device_get_child_node_count(dev);
176     if (!count)
177         return ERR_PTR(-ENODEV);
178 
179     priv = devm_kzalloc(dev, sizeof_gpio_leds_priv(count),
                 GFP_KERNEL);
180     if (!priv)
181         return ERR_PTR(-ENOMEM);
182 
183     device_for_each_child_node(dev, child) {
184         struct gpio_led led = {};
185         const char *state = NULL;
186 
187         led.gpiod = devm_get_gpiod_from_child(dev, NULL, child);
188         if (IS_ERR(led.gpiod)) {
189             fwnode_handle_put(child);
190             ret = PTR_ERR(led.gpiod);
191             goto err;
192         }
193 
194         np = of_node(child);
195 
196         if (fwnode_property_present(child, "label")) {
197             fwnode_property_read_string(child, "label", &led.name);
198         } else {
199             if (IS_ENABLED(CONFIG_OF) && !led.name && np)
200                 led.name = np->name;
201             if (!led.name)
202                 return ERR_PTR(-EINVAL);
203         }
```

```
204         fwnode_property_read_string(child, "linux,default-trigger",
205                     &led.default_trigger);
206 
207         if (!fwnode_property_read_string(child, "default-state",
208                     &state)) {
209             if (!strcmp(state, "keep"))
210                 led.default_state = LEDS_GPIO_DEFSTATE_KEEP;
211             else if (!strcmp(state, "on"))
212                 led.default_state = LEDS_GPIO_DEFSTATE_ON;
213             else
214                 led.default_state = LEDS_GPIO_DEFSTATE_OFF;
215         }
216 
217         if (fwnode_property_present(child, "retain-state-suspended"))
218             led.retain_state_suspended = 1;
219 
220         ret = create_gpio_led(&led, &priv->leds[priv->num_leds++],
221                   dev, NULL);
222         if (ret < 0) {
223             fwnode_handle_put(child);
224             goto err;
225         }
226     }
227 
228     return priv;
229 
230 err:
231     for (count = priv->num_leds - 2; count >= 0; count--)
232         delete_gpio_led(&priv->leds[count]);
233     return ERR_PTR(ret);
234 }
```

第 175 行，调用 device_get_child_node_count 函数统计子节点数量，一般在在设备树中创建一个节点表示 LED 灯，然后在这个节点下面为每个 LED 灯创建一个子节点。因此子节点数量也是 LED 灯的数量。

第 183 行，遍历每个子节点，获取每个子节点的信息。

第 187 行，获取 LED 灯所使用的 GPIO 信息。

第 196~197 行，读取子节点 label 属性值，因为使用 label 属性作为 LED 的名字。

第 204~205 行，获取“linux,default-trigger”属性值，可以通过此属性设置某个 LED 灯在 Linux 系统中的默认功能，比如作为系统心跳指示灯等等。

第 207~215 行，获取“default-state”属性值，也就是 LED 灯的默认状态属性。

第 220 行，调用 create_gpio_led 函数创建 LED 相关的 io，其实就是设置 LED 所使用的 io 为输出之类的。create_gpio_led 函数主要是初始化 led_dat 这个 gpio_led_data 结构体类型变量，led_dat 保存了 LED 的操作函数等内容。

关于 gpio_led_probe 函数就分析到这里，gpio_led_probe 函数主要功能就是获取 LED 灯的设备信息，然后根据这些信息来初始化对应的 IO，设置为输出等。

56.3 设备树节点编写

打开文档 Documentation/devicetree/bindings/leds/leds-gpio.txt，此文档详细的讲解了 Linux 自带驱动对应的设备树节点该如何编写，我们在编写设备节点的时候要注意一下几点：

①、创建一个节点表示 LED 灯设备，比如 dtsleds，如果板子上有多个 LED 灯的话每个 LED 灯都作为 dtsleds 的子节点。

②、dtsleds 节点的 compatible 属性值一定要为“gpio-leds”。

③、设置 label 属性，此属性为可选，每个子节点都有一个 label 属性，label 属性一般表示 LED 灯的名字，比如以颜色区分的话就是 red、green 等等。

④、每个子节点必须要设置 gpios 属性值，表示此 LED 所使用的 GPIO 引脚！

⑤、可以设置“linux,default-trigger”属性值，也就是设置 LED 灯的默认功能，可以查阅 Documentation/devicetree/bindings/leds/common.txt 这个文档来查看可选功能，比如：

backlight：LED 灯作为背光。

default-on：LED 灯打开

heartbeat：LED 灯作为心跳指示灯，可以作为系统运行提示灯。

ide-disk：LED 灯作为硬盘活动指示灯。

timer：LED 灯周期性闪烁，由定时器驱动，闪烁频率可以修改

⑥、可以设置“default-state”属性值，可以设置为 on、off 或 keep，为 on 的时候 LED 灯默认打开，为 off 的话 LED 灯默认关闭，为 keep 的话 LED 灯保持当前模式。

根据上述几条要求在 imx6ull-alientek-emmc.dts 中添加如下所示 LED 灯设备节点：

示例代码 56.3.1 dtsleds 设备节点

```
1  dtsleds {
2      compatible = "gpio-leds";
3  
4      led0 {
5          label = "red";
6          gpios = <&gpio1 3 GPIO_ACTIVE_LOW>;
7          default-state = "off";
8      };
9  };
```

因为 I.MX6U-ALPHA 开发板只有一个 LED0，因此在 dtsleds 这个节点下只有一个子节点 led0，LED0 名字为 red，默认关闭。修改完成以后保存并重新编译设备树，然后用新的设备树启动开发板。

56.4 运行测试

用新的 zImage 和 imx6ull-alientek-emmc.dtb 启动开发板，启动以后查看 /sys/bus/platform/devices/dtsleds 这个目录是否存在，如果存在的话就如到此目录中，如图 56.4.1 所示：

```
/sys/devices/platform/dtsleds # ls
driver            leds              of_node           subsystem
driver_override   modalias          power             uevent
```

图 56.4.1 dtsleds 目录

进入到 leds 目录中，此目录中的内容如图 56.4.2 所示：

```
/sys/devices/platform/dtsleds/leds # ls
red
/sys/devices/platform/dtsleds/leds #
```

图 56.4.2 leds 目录内容

从图 56.4.2 可以看出，在 leds 目录下有一个名为“red”子目录，这个子目录的名字就是我们在设备树中第 5 行设置的 label 属性值。

我们的设置究竟有没有用，最终是要通过测试才能知道的，首先查看一下系统中有没有“sys/class/leds/red/brightness”这个文件，如果有的话就输入如下命令打开 RED 这个 LED 灯：

```
echo 1 > sys/class/leds/red/brightness          //打开 LED0
```

关闭 RED 这个 LED 灯的命令如下：

```
echo 0 > sys/class/leds/red/brightness          //关闭 LED0
```

如果能正常的打开和关闭 LED 灯话就说明我们 Linux 内核自带的 LED 灯驱动工作正常。我们一般会使用一个 LED 灯作为系统指示灯，系统运行正常的话这个 LED 指示灯就会一闪一闪的。里我们设置 LED0 作为系统指示灯，在 dtsleds 这个设备节点中加入“linux,default-trigger”属性信息即可，属性值为“heartbeat”，修改完以后的 dtsleds 节点内容如下：

示例代码 56.3.2 dtsleds 设备节点

```
1  dtsleds {
2      compatible = "gpio-leds";
3  
4      led0 {
5          label = "red";
6          gpios = <&gpio1 3 GPIO_ACTIVE_LOW>;
7          linux,default-trigger = "heartbeat";
8          default-state = "on";
9      };
10 };
```

第 7 行，设置 LED0 为 heartbeat。

第 8 行，默认打开 LED0。

重新编译设备树并且使用新的设备树启动 Linux 系统，启动以后 LED0 就会闪烁，作为系统心跳指示灯，表示系统正在运行。

第五十七章 Linux MISC 驱动实验

misc 的意思是混合、杂项的，因此 MISC 驱动也叫做杂项驱动，也就是当我们板子上的某些外设无法进行分类的时候就可以使用 MISC 驱动。MISC 驱动其实就是最简单的字符设备驱动，通常嵌套在 platform 总线驱动中，实现复杂的驱动，本章我们就来学习一下 MISC 驱动的编写。

57.1 MISC 设备驱动简介

所有的 MISC 设备驱动的主设备号都为 10，不同的设备使用不同的从设备号。随着 Linux 字符设备驱动的不断增加，设备号变得越来越紧张，尤其是主主设备号，MISC 设备驱动就用于解决此问题。MISC 设备会自动创建 cdev，不需要像我们以前那样手动创建，因此采用 MISC 设备驱动可以简化字符设备驱动的编写。我们需要向 Linux 注册一个 miscdevice 设备，miscdevice 是一个结构体，定义在文件

示例代码 57.1.1 miscdevice 结构体代码

```
57 struct miscdevice  {
58     int minor;                                  /* 子设备号    */
59     const char *name;                           /* 设备名字    */
60     const struct file_operations *fops;        /* 设备操作集  */
61     struct list_head list;
62     struct device *parent;
63     struct device *this_device;
64     const struct attribute_group **groups;
65     const char *nodename;
66     umode_t mode;
67 };
```

定义一个 MISC 设备(miscdevice 类型)以后我们需要设置 minor、name 和 fops 这三个成员变量。minor 表示子设备号，MISC 设备的主设备号为 10，这个是固定的，需要用户指定子设备号，Linux 系统已经预定义了一些 MISC 设备的子设备号，这些预定义的子设备号定义在 include/linux/miscdevice.h 文件中，如下所示：

示例代码 57.1.2 预定义的 MISC 设备子设备号

```
13 #define PSMOUSE_MINOR           1
14 #define MS_BUSMOUSE_MINOR       2   /* unused */
15 #define ATIXL_BUSMOUSE_MINOR    3   /* unused */
16 /*#define AMIGAMOUSE_MINOR      4   FIXME OBSOLETE */
17 #define ATARIMOUSE_MINOR        5   /* unused */
18 #define SUN_MOUSE_MINOR         6   /* unused */
......
52 #define MISC_DYNAMIC_MINOR      255
```

我们在使用的时候可以从这些预定义的子设备号中挑选一个，当然也可以自己定义，只要这个子设备号没有被其他设备使用接口。

name 就是此 MISC 设备名字，当此设备注册成功以后就会在/dev 目录下生成一个名为 name 的设备文件。fops 就是字符设备的操作集合，MISC 设备驱动最终是需要使用用户提供的 fops 操作集合。

当设置好 miscdevice 以后就需要使用 misc_register 函数向系统中注册一个 MISC 设备，此函数原型如下：

```
int misc_register(struct miscdevice * misc)
```

函数参数和返回值含义如下：

misc：要注册的 MISC 设备。

返回值：负数，失败；0，成功。

以前我们需要自己调用一堆的函数去创建设备，比如在以前的字符设备驱动中我们会使用如下几个函数完成设备创建过程：

示例代码 57.1.3 传统的创建设备过程

```
1 alloc_chrdev_region();     /* 申请设备号   */
2 cdev_init();               /* 初始化 cdev  */
3 cdev_add();                /* 添加 cdev    */
4 class_create();            /* 创建类       */
5 device_create();           /* 创建设备     */
```

现在我们可以直接使用 misc_register 一个函数来完成示例代码 57.1.2 中的这些步骤。当我们卸载设备驱动模块的时候需要调用 misc_deregister 函数来注销掉 MISC 设备，函数原型如下：

```
int misc_deregister(struct miscdevice *misc)
```

函数参数和返回值含义如下：

misc：要注销的 MISC 设备。

返回值：负数，失败；0，成功。

以前注销设备驱动的时候，我们需要调用一堆的函数去删除此前创建的 cdev、设备等等内容，如下所示：

示例代码 57.1.3 传统的删除设备的过程

```
1 cdev_del();                    /*  删除 cdev   */
2 unregister_chrdev_region();    /* 注销设备号   */
3 device_destroy();              /* 删除设备     */
4 class_destroy();               /* 删除类       */
```

现在我们只需要一个 misc_deregister 函数即可完成示例代码 57.1.3 中的这些工作。关于 MISC 设备驱动就讲解到这里，接下来我们就使用 platform 加 MISC 驱动框架来编写 beep 蜂鸣器驱动。

57.2 硬件原理图分析

本章实验我们只使用到 IMX6U-ALPHA 开发板上的 BEEP 蜂鸣器，因此实验硬件原理图参考 14.3 小节即可。

57.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 17_misc。

本章实验我们采用 platform 加 misc 的方式编写 beep 驱动，这也是实际的 Linux 驱动中很常用的方法。采用 platform 来实现总线、设备和驱动，misc 主要负责完成字符设备的创建。

57.3.1 修改设备树

本章实验我们需要用到蜂鸣器，因此需要在 imx6ull-alientek-emmc.dts 文件中创建蜂鸣器设备节点，这里我们直接使用 46.3.1 小节创建的 beep 这个设备节点即可。

57.3.2 beep 驱动程序编写

新建名为“19_miscbeep”的文件夹，然后在 19_miscbeep 文件夹里面创建 vscode 工程，工作区命名为“miscbeep。新建名为 miscbeep.c 的驱动文件，在 miscbeep.c 中输入如下所示内容：

示例代码 57.3.2.1 miscbeep.c 文件代码段

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of.h>
12  #include <linux/of_address.h>
13  #include <linux/of_gpio.h>
14  #include <linux/platform_device.h>
15  #include <linux/miscdevice.h>
16  #include <asm/mach/map.h>
17  #include <asm/uaccess.h>
18  #include <asm/io.h>
19  /***************************************************************
20  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
21  文件名      : miscbeep.c
22  作者        : 左忠凯
23  版本        : V1.0
24  描述        : 采用 MISC 的蜂鸣器驱动程序。
25  其他        : 无
26  论坛        : www.openedv.com
27  日志        : 初版 V1.0 2019/8/20 左忠凯创建
28  ***************************************************************/
29  #define MISCBEEP_NAME    "miscbeep"     /* 名字     */
30  #define MISCBEEP_MINOR   144            /* 子设备号 */
31  #define BEEPOFF          0              /* 关蜂鸣器 */
32  #define BEEPON           1              /* 开蜂鸣器 */
33  
34  /* miscbeep 设备结构体 */
35  struct miscbeep_dev{
36      dev_t devid;                /* 设备号               */
37      struct cdev cdev;           /* cdev                 */
38      struct class *class;        /* 类                   */
39      struct device *device;      /* 设备                 */
40      struct device_node  *nd;    /* 设备节点             */
41      int beep_gpio;              /* beep 所使用的 GPIO 编号  */
42  };
43  
```

```
44 struct miscbeep_dev miscbeep;   /* beep 设备           */
45
46 /*
47  * @description  : 打开设备
48  * @param – inode : 传递给驱动的 inode
49  * @param - filp : 设备文件，file 结构体有个叫做 private_data 的成员变量
50  *                 一般在 open 的时候将 private_data 指向设备结构体。
51  * @return        : 0 成功;其他 失败
52  */
53 static int miscbeep_open(struct inode *inode, struct file *filp)
54 {
55     filp->private_data = &miscbeep; /* 设置私有数据 */
56     return 0;
57 }
58
59 /*
60  * @description  : 向设备写数据
61  * @param - filp : 设备文件，表示打开的文件描述符
62  * @param - buf  : 要写给设备写入的数据
63  * @param - cnt  : 要写入的数据长度
64  * @param - offt : 相对于文件首地址的偏移
65  * @return       : 写入的字节数，如果为负值，表示写入失败
66  */
67 static ssize_t miscbeep_write(struct file *filp,
                   const char __user *buf, size_t cnt, loff_t *offt)
68 {
69     int retvalue;
70     unsigned char databuf[1];
71     unsigned char beepstat;
72     struct miscbeep_dev *dev = filp->private_data;
73
74     retvalue = copy_from_user(databuf, buf, cnt);
75     if(retvalue < 0) {
76         printk("kernel write failed!\r\n");
77         return -EFAULT;
78     }
79
80     beepstat = databuf[0];              /* 获取状态值   */
81     if(beepstat == BEEPON) {
82         gpio_set_value(dev->beep_gpio, 0); /* 打开蜂鸣器   */
83     } else if(beepstat == BEEPOFF) {
84         gpio_set_value(dev->beep_gpio, 1); /* 关闭蜂鸣器   */
85     }
```

```
86      return 0;
87  }
88
89  /* 设备操作函数 */
90  static struct file_operations miscbeep_fops = {
91      .owner = THIS_MODULE,
92      .open = miscbeep_open,
93      .write = miscbeep_write,
94  };
95
96  /* MISC 设备结构体 */
97  static struct miscdevice beep_miscdev = {
98      .minor = MISCBEEP_MINOR,
99      .name = MISCBEEP_NAME,
100     .fops = &miscbeep_fops,
101 };
102
103  /*
104   * @description  : flatform 驱动的 probe 函数，当驱动与
105   *                 设备匹配以后此函数就会执行
106   * @param - dev  : platform 设备
107   * @return       : 0，成功;其他负值,失败
108   */
109 static int miscbeep_probe(struct platform_device *dev)
110 {
111     int ret = 0;
112
113     printk("beep driver and device was matched!\r\n");
114     /* 设置 BEEP 所使用的 GPIO */
115     /* 1、获取设备节点：beep */
116     miscbeep.nd = of_find_node_by_path("/beep");
117     if(miscbeep.nd == NULL) {
118         printk("beep node not find!\r\n");
119         return -EINVAL;
120     }
121
122     /* 2、 获取设备树中的 gpio 属性，得到 BEEP 所使用的 BEEP 编号 */
123     miscbeep.beep_gpio = of_get_named_gpio(miscbeep.nd, "beep-gpio",
                                               0);
124     if(miscbeep.beep_gpio < 0) {
125         printk("can't get beep-gpio");
126         return -EINVAL;
127     }
```

```
128 
129     /* 3、设置 GPIO5_IO01 为输出，并且输出高电平，默认关闭 BEEP */
130     ret = gpio_direction_output(miscbeep.beep_gpio, 1);
131     if(ret < 0) {
132         printk("can't set gpio!\r\n");
133     }
134 
135     /* 一般情况下会注册对应的字符设备，但是这里我们使用 MISC 设备
136      * 所以我们不需要自己注册字符设备驱动，只需要注册 misc 设备驱动即可
137      */
138     ret = misc_register(&beep_miscdev);
139     if(ret < 0){
140         printk("misc device register failed!\r\n");
141         return -EFAULT;
142     }
143 
144     return 0;
145 }
146 
147 /*
148  * @description   : remove 函数，移除 platform 驱动的时候此函数会执行
149  * @param - dev   : platform 设备
150  * @return        : 0，成功;其他负值,失败
151  */
152 static int miscbeep_remove(struct platform_device *dev)
153 {
154     /* 注销设备的时候关闭 LED 灯 */
155     gpio_set_value(miscbeep.beep_gpio, 1);
156 
157     /* 注销 misc 设备驱动 */
158     misc_deregister(&beep_miscdev);
159     return 0;
160 }
161 
162  /* 匹配列表 */
163  static const struct of_device_id beep_of_match[] = {
164      { .compatible = "atkalpha-beep" },
165      { /* Sentinel */ }
166  };
167 
168  /* platform 驱动结构体 */
169 static struct platform_driver beep_driver = {
170      .driver     = {
```

```
171         .name   = "imx6ul-beep",          /* 驱动名字     */
172         .of_match_table = beep_of_match,  /* 设备树匹配表 */
173     },
174     .probe      = miscbeep_probe,
175     .remove     = miscbeep_remove,
176 };
177 
178 /*
179  * @description : 驱动入口函数
180  * @param          : 无
181  * @return         : 无
182  */
183 static int __init miscbeep_init(void)
184 {
185     return platform_driver_register(&beep_driver);
186 }
187 
188 /*
189  * @description   : 驱动出口函数
190  * @param          : 无
191  * @return         : 无
192  */
193 static void __exit miscbeep_exit(void)
194 {
195     platform_driver_unregister(&beep_driver);
196 }
197 
198 module_init(miscbeep_init);
199 module_exit(miscbeep_exit);
200 MODULE_LICENSE("GPL");
201 MODULE_AUTHOR("zuozhongkai");
```

第 29~94 行，标准的字符设备驱动。

第 97~101 行，MISC 设备 beep_miscdev，第 98 行设置子设备号为 144，第 99 行设置设备名字为“miscbeep”，这样当系统启动以后就会在/dev/目录下存在一个名为“miscbeep”的设备文件。第 100 行，设置 MISC 设备的操作函数集合，为 file_operations 类型。

第 109~145 行，platform 框架的 probe 函数，当驱动与设备匹配以后此函数就会执行，首先在此函数中初始化 BEEP 所使用的 IO。最后在 138 行通过 misc_register 函数向 Linux 内核注册 MISC 设备，也就是前面定义的 beep_miscdev。

第 152~160 行，platform 框架的 remove 函数，在此函数中调用 misc_deregister 函数来注销 MISC 设备。

第 163~196，标准的 platform 驱动。

57.3.3 编写测试 APP

新建 miscbeepApp.c 文件，然后在里面输入如下所示内容：

示例代码 57.3.2.2 miscbeepApp.c 文件代码段

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "fcntl.h"
6  #include "stdlib.h"
7  #include "string.h"
8  /***************************************************************
9  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
10 文件名        : miscbeepApp.c
11 作者          : 左忠凯
12 版本          : V1.0
13 描述          : MISC 驱动框架下的 beep 测试 APP。
14 其他          : 无
15 使用方法      : ./miscbeepApp  /dev/miscbeep  0 关闭蜂鸣器
16                 ./misdcbeepApp /dev/miscbeep  1 打开蜂鸣器
17 论坛          : www.openedv.com
18 日志          : 初版 V1.0 2019/8/20 左忠凯创建
19 ***************************************************************/
20 #define BEEPOFF  0
21 #define BEEPON   1
22 
23 /*
24  * @description    : main 主程序
25  * @param - argc   : argv 数组元素个数
26  * @param - argv   : 具体参数
27  * @return         : 0 成功;其他 失败
28  */
29 int main(int argc, char *argv[])
30 {
31     int fd, retvalue;
32     char *filename;
33     unsigned char databuf[1];
34 
35     if(argc != 3){
36         printf("Error Usage!\r\n");
37         return -1;
38     }
39 
```

```
40      filename = argv[1];
41      fd = open(filename, O_RDWR);    /* 打开 beep 驱动 */
42      if(fd < 0){
43          printf("file %s open failed!\r\n", argv[1]);
44          return -1;
45      }
46
47      databuf[0] = atoi(argv[2]);     /* 要执行的操作：打开或关闭 */
48      retvalue = write(fd, databuf, sizeof(databuf));
49      if(retvalue < 0){
50          printf("BEEP Control Failed!\r\n");
51          close(fd);
52          return -1;
53      }
54
55      retvalue = close(fd);           /* 关闭文件 */
56      if(retvalue < 0){
57          printf("file %s close failed!\r\n", argv[1]);
58          return -1;
59      }
60      return 0;
61  }
```

miscbeepApp.c 文件内容和其他例程的测试 APP 基本一致，很简单，这里就不讲解了。

57.4 运行测试

57.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为“leddevice.o leddriver.o”，Makefile 内容如下所示：

示例代码 57.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := miscbeep.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“miscbeep.o”。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“miscbeep.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 miscbeepApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc miscbeepApp.c -o miscbeepApp
```

编译成功以后就会生成 miscbeepApp 这个应用程序。

57.4.2 运行测试

将上一小节编译出来 miscbeep.ko 和 miscbeepApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中，输入如下命令加载 miscbeep.ko 这个驱动模块。

```
depmod                //第一次加载驱动的时候需要运行此命令
modprobe miscbeep.ko  //加载设备模块
```

当驱动模块加载成功以后我们可以在/sys/class/misc 这个目录下看到一个名为"miscbeep"的子目录，如图 57.4.2.1 所示：

```
/lib/modules/4.1.15 # ls /sys/class/misc/
autofs            memory_bandwidth    rfkill
cpu_dma_latency   miscbeep            ubi_ctrl
fuse              mxc_asrc            watchdog
hw_random         network_latency
loop-control      network_throughput
/lib/modules/4.1.15 #
```

图 57.4.2.1 misbeep 子目录

所有的 misc 设备都属于同一个类，/sys/class/misc 目录下就是 misc 这个类的所有设备，每个设备对应一个子目录。

驱动与设备匹配成功以后就会生成/dev/miscbeep 这个设备驱动文件，输入如下命令查看这个文件的主次设备号：

```
ls /dev/miscbeep -l
```

结果如图 57.4.2.2 所示：

```
/lib/modules/4.1.15 # ls /dev/miscbeep -l
crw-rw----    1 0        0          10, 144 Jan  1 05:07 /dev/miscbeep
/lib/modules/4.1.15 #
```

图 57.4.2.2 /dev/miscbeep 设备文件

从图 57.4.2.2 可以看出，/dev/miscbeep 这个设备的主设备号为 10，次设备号为 144，和我们驱动程序里面设置的一致。

输入如下命令打开 BEEP：

```
./miscbeepApp /dev/miscbeep 1    //打开 BEEP
```

在输入如下命令关闭 LED 灯：

```
./miscbeepApp /dev/miscbeep 0    //关闭 BEEP
```

观察一下 BEEP 能否打开和关闭，如果可以的话就说明驱动工作正常，如果要卸载驱动的话输入如下命令即可：

```
rmmod miscbeep.ko
```

第五十八章 Linux INPUT 子系统实验

按键、鼠标、键盘、触摸屏等都属于输入(input)设备，Linux 内核为此专门做了一个叫做 input 子系统的框架来处理输入事件。输入设备本质上还是字符设备，只是在此基础上套上了 input 框架，用户只需要负责上报输入事件，比如按键值、坐标等信息，input 核心层负责处理这些事件。本章我们就来学习一下 Linux 内核中的 input 子系统。

58.1 input 子系统

58.1.1 input 子系统简介

input 就是输入的意思，因此 input 子系统就是管理输入的子系统，和 pinctrl 和 gpio 子系统一样，都是 Linux 内核针对某一类设备而创建的框架。比如按键输入、键盘、鼠标、触摸屏等等这些都属于输入设备，不同的输入设备所代表的含义不同，按键和键盘就是代表按键信息，鼠标和触摸屏代表坐标信息，因此在应用层的处理就不同，对于驱动编写者而言不需要去关心应用层的事情，我们只需要按照要求上报这些输入事件即可。为此 input 子系统分为 input 驱动层、input 核心层、input 事件处理层，最终给用户空间提供可访问的设备节点，input 子系统框架如图 58.1.1.1 所示：

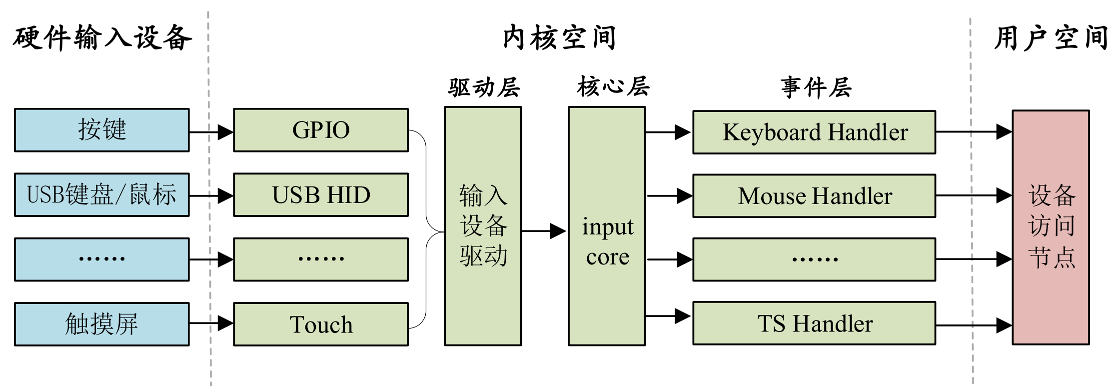


图 58.1.1.1 input 子系统结构图

图 58.1.1 中左边就是最底层的具体设备，比如按键、USB 键盘/鼠标等，中间部分属于 Linux 内核空间，分为驱动层、核心层和时间层，最右边的就是用户空间，所有的输入设备以文件的形式供用户应用程序使用。可以看出 input 子系统用到了我们前面讲解的驱动分层模型，我们编写驱动程序的时候只需要关注中间的驱动层、核心层和事件层，这三个层的分工如下：

驱动层：输入设备的具体驱动程序，比如按键驱动程序，向内核层报告输入内容。

核心层：承上启下，为驱动层提供输入设备注册和操作接口。通知事件层对输入事件进行处理。

事件层：主要和用户空间进行交互。

58.1.2 input 驱动编写流程

input 核心层会向 Linux 内核注册一个字符设备，大家找到 drivers/input/input.c 这个文件，input.c 就是 input 输入子系统的核心层，此文件里面有如下所示代码：

示例代码 58.1.2.1 input 核心层创建字符设备过程

```
1767 struct class input_class = {
1768    .name       = "input",
1769    .devnode    = input_devnode,
1770 };
......
2414 static int __init input_init(void)
2415 {
2416    int err;
```

```
2417
2418    err = class_register(&input_class);
2419    if (err) {
2420        pr_err("unable to register input_dev class\n");
2421        return err;
2422    }
2423
2424    err = input_proc_init();
2425    if (err)
2426        goto fail1;
2427
2428    err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
2429                    INPUT_MAX_CHAR_DEVICES, "input");
2430    if (err) {
2431        pr_err("unable to register char major %d", INPUT_MAJOR);
2432        goto fail2;
2433    }
2434
2435    return 0;
2436
2437  fail2:    input_proc_exit();
2438  fail1:    class_unregister(&input_class);
2439    return err;
2440 }
```

第 2418 行，注册一个 input 类，这样系统启动以后就会在/sys/class 目录下有一个 input 子目录，如图 58.1.2.1 所示：

```
/ # ls sys/class/
ata_device     gpio          misc           regulator      tty
ata_link       graphics      mmc_host       rfkill         ubi
ata_port       i2c-dev       mtd            rtc            udc
backlight      ieee80211     net            scsi_device    vc
bdi            input         power_supply   scsi_disk      video4linux
block          lcd           pps            scsi_host      vtconsole
dma            leds          ptp            sound          watchdog
drm            mdio_bus      pwm            spi_master
firmware       mem           rc             thermal
/ #
```

图 58.1.2.1 input 类

第 2428~2429 行，注册一个字符设备，主设备号为 INPUT_MAJOR，INPUT_MAJOR 定义在 include/uapi/linux/major.h 文件中，定义如下：

```
#define INPUT_MAJOR            13
```

因此，input 子系统的所有设备主设备号都为 13，我们在使用 input 子系统处理输入设备的时候就不需要去注册字符设备了，我们只需要向系统注册一个 input_device 即可。

1、注册 input_dev

在使用 input 子系统的时候我们只需要注册一个 input 设备即可，input_dev 结构体表示 input 设备，此结构体定义在 include/linux/input.h 文件中，定义如下(有省略)：

示例代码 58.1.2.2 input_dev 结构体

```
121 struct input_dev {
122     const char *name;
123     const char *phys;
124     const char *uniq;
125     struct input_id id;
126
127     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];
128
129     unsigned long evbit[BITS_TO_LONGS(EV_CNT)];   /* 事件类型的位图 */
130     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值的位图   */
131     unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标的位图 */
132     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标的位图 */
133     unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件的位图 */
134     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /*LED 相关的位图   */
135     unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];/* sound 有关的位图 */
136     unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];   /* 压力反馈的位图 */
137     unsigned long swbit[BITS_TO_LONGS(SW_CNT)];   /*开关状态的位图   */
......
189     bool devres_managed;
190 };
```

第 129 行，evbit 表示输入事件类型，可选的事件类型定义在 include/uapi/linux/input.h 文件中，事件类型如下：

示例代码 58.1.2.3 事件类型

```
#define EV_SYN          0x00    /* 同步事件          */
#define EV_KEY          0x01    /* 按键事件          */
#define EV_REL          0x02    /* 相对坐标事件      */
#define EV_ABS          0x03    /* 绝对坐标事件      */
#define EV_MSC          0x04    /* 杂项(其他)事件    */
#define EV_SW           0x05    /* 开关事件          */
#define EV_LED          0x11    /* LED               */
#define EV_SND          0x12    /* sound(声音)       */
#define EV_REP          0x14    /* 重复事件          */
#define EV_FF           0x15    /* 压力事件          */
#define EV_PWR          0x16    /* 电源事件          */
#define EV_FF_STATUS    0x17    /* 压力状态事件      */
```

比如本章我们要使用到按键，那么就需要注册 EV_KEY 事件，如果要使用连按功能的话还需要注册 EV_REP 事件。

继续回到示例代码 58.1.2.2 中，第 129 行~137 行的 evbit、keybit、relbit 等等都是存放不同事件对应的值。比如我们本章要使用按键事件，因此要用到 keybit，keybit 就是按键事件使用的位图，Linux 内核定义了很多按键值，这些按键值定义在 include/uapi/linux/input.h 文件中，按键值如下：

示例代码 58.1.2.4 按键值

```
215 #define KEY_RESERVED        0
216 #define KEY_ESC             1
217 #define KEY_1               2
218 #define KEY_2               3
219 #define KEY_3               4
220 #define KEY_4               5
221 #define KEY_5               6
222 #define KEY_6               7
223 #define KEY_7               8
224 #define KEY_8               9
225 #define KEY_9               10
226 #define KEY_0               11
......
794 #define BTN_TRIGGER_HAPPY39 0x2e6
795 #define BTN_TRIGGER_HAPPY40 0x2e7
```

我们可以将开发板上的按键值设置为示例代码 58.1.2.4 中的任意要一个，比如我们本章实验会将 I.MX6U-ALPHA 开发板上的 KEY 按键值设置为 KEY_0。

在编写 input 设备驱动的时候我们需要先申请一个 input_dev 结构体变量，使用 input_allocate_device 函数来申请一个 input_dev，此函数原型如下所示：

```
struct input_dev *input_allocate_device(void)
```

函数参数和返回值含义如下：

参数：无。

返回值：申请到的 input_dev。

如果要注销的 input 设备的话需要使用 input_free_device 函数来释放掉前面申请到的 input_dev，input_free_device 函数原型如下：

```
void input_free_device(struct input_dev *dev)
```

函数参数和返回值含义如下：

dev：需要释放的 input_dev。

返回值：无。

申请好一个 input_dev 以后就需要初始化这个 input_dev，需要初始化的内容主要为事件类型(evbit)和事件值(keybit)这两种。input_dev 初始化完成以后就需要向 Linux 内核注册 input_dev 了，需要用到 input_register_device 函数，此函数原型如下：

```
int input_register_device(struct input_dev *dev)
```

函数参数和返回值含义如下：

dev：要注册的 input_dev 。

返回值：0，input_dev 注册成功；负值，input_dev 注册失败。

同样的，注销 input 驱动的时候也需要使用 input_unregister_device 函数来注销掉前面注册的 input_dev，input_unregister_device 函数原型如下：

```
void input_unregister_device(struct input_dev *dev)
```

函数参数和返回值含义如下：

dev：要注销的 input_dev 。

返回值：无。

综上所述，input_dev 注册过程如下：

①、使用 input_allocate_device 函数申请一个 input_dev。

②、初始化 input_dev 的事件类型以及事件值。

③、使用 input_unregister_device 函数向 Linux 系统注册前面初始化好的 input_dev。

④、卸载 input 驱动的时候需要先使用 input_unregister_device 函数注销掉注册的 input_dev，然后使用 input_free_device 函数释放掉前面申请的 input_dev。input_dev 注册过程示例代码如下所示：

示例代码 58.1.2.5 input_dev 注册流程

```
1  struct input_dev *inputdev;      /* input 结构体变量 */
2  
3  /* 驱动入口函数 */
4  static int __init xxx_init(void)
5  {
6      ......
7      inputdev = input_allocate_device();    /* 申请 input_dev      */
8      inputdev->name = "test_inputdev";      /* 设置 input_dev 名字  */
9  
10     /*********第一种设置事件和事件值的方法***********/
11     __set_bit(EV_KEY, inputdev->evbit);    /* 设置产生按键事件      */
12     __set_bit(EV_REP, inputdev->evbit);    /* 重复事件             */
13     __set_bit(KEY_0, inputdev->keybit);    /*设置产生哪些按键值     */
14     /************************************************/
15 
16     /*********第二种设置事件和事件值的方法***********/
17     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | 
                                        BIT_MASK(EV_REP);
18     keyinputdev.inputdev->keybit[BIT_WORD(KEY_0)] |= 
                                        BIT_MASK(KEY_0);
19     /************************************************/
20 
21     /*********第三种设置事件和事件值的方法***********/
22     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | 
                                      BIT_MASK(EV_REP);
23     input_set_capability(keyinputdev.inputdev, EV_KEY, KEY_0);
24     /************************************************/
25 
26     /* 注册 input_dev */
27     input_register_device(inputdev);
28     ......
29     return 0;
30 }
31 
32 /* 驱动出口函数 */
33 static void __exit xxx_exit(void)
```

```
34 {
35     input_unregister_device(inputdev);      /* 注销 input_dev  */
36     input_free_device(inputdev);            /* 删除 input_dev  */
37 }
```

第 1 行，定义一个 input_dev 结构体指针变量。

第 4~30 行，驱动入口函数，在此函数中完成 input_dev 的申请、设置、注册等工作。第 7 行调用 input_allocate_device 函数申请一个 input_dev。第 10~23 行都是设置 input 设备事件和按键值，这里用了三种方法来设置事件和按键值。第 27 行调用 input_register_device 函数向 Linux 内核注册 inputdev。

第 33~37 行，驱动出口函数，第 35 行调用 input_unregister_device 函数注销前面注册的 input_dev，第 36 行调用 input_free_device 函数删除前面申请的 input_dev。

2、上报输入事件

当我们向 Linux 内核注册好 input_dev 以后还不能高枕无忧的使用 input 设备，input 设备都是具有输入功能的，但是具体是什么样的输入值 Linux 内核是不知道的，我们需要获取到具体的输入值，或者说是输入事件，然后将输入事件上报给 Linux 内核。比如按键，我们需要在按键中断处理函数，或者消抖定时器中断函数中将按键值上报给 Linux 内核，这样 Linux 内核才能获取到正确的输入值。不同的事件，其上报事件的 API 函数不同，我们依次来看一下一些常用的事件上报 API 函数。

首先是 input_event 函数，此函数用于上报指定的事件以及对应的值，函数原型如下：

```
void input_event(struct input_dev        *dev,
                 unsigned int            type,
                 unsigned int            code,
                 int                     value)
```

函数参数和返回值含义如下：

dev：需要上报的 input_dev。

type: 上报的事件类型，比如 EV_KEY。

code：事件码，也就是我们注册的按键值，比如 KEY_0、KEY_1 等等。

value：事件值，比如 1 表示按键按下，0 表示按键松开。

返回值：无。

input_event 函数可以上报所有的事件类型和事件值，Linux 内核也提供了其他的针对具体事件的上报函数，这些函数其实都用到了 input_event 函数。比如上报按键所使用的 input_report_key 函数，此函数内容如下：

例代码 58.1.2.6 input_report_key 函数

```
static inline void input_report_key(struct input_dev *dev,
                                    unsigned int code, int value)
{
    input_event(dev, EV_KEY, code, !!value);
}
```

从示例代码 58.1.2.6 可以看出，input_report_key 函数的本质就是 input_event 函数，如果要上报按键事件的话还是建议大家使用 input_report_key 函数。

同样的还有一些其他的事件上报函数，这些函数如下所示：

```
void input_report_rel(struct input_dev *dev, unsigned int code, int value)
void input_report_abs(struct input_dev *dev, unsigned int code, int value)
```

```
void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
void input_report_switch(struct input_dev *dev, unsigned int code, int value)
void input_mt_sync(struct input_dev *dev)
```

当我们上报事件以后还需要使用 input_sync 函数来告诉 Linux 内核 input 子系统上报结束，input_sync 函数本质是上报一个同步事件，此函数原型如下所示：

```
void input_sync(struct input_dev *dev)
```

函数参数和返回值含义如下：

dev：需要上报同步事件的 input_dev。

返回值：无。

综上所述，按键的上报事件的参考代码如下所示：

示例代码 58.1.2.7 事件上报参考代码

```
1  /* 用于按键消抖的定时器服务函数 */
2  void timer_function(unsigned long arg)
3  {
4      unsigned char value;
5  
6      value = gpio_get_value(keydesc->gpio);    /* 读取 IO 值    */
7      if(value == 0){                           /* 按下按键      */
8          /* 上报按键值 */
9          input_report_key(inputdev, KEY_0, 1); /* 最后一个参数 1，按下 */
10         input_sync(inputdev);                 /*  同步事件     */
11     } else {                                  /* 按键松开      */
12         input_report_key(inputdev, KEY_0, 0); /* 最后一个参数 0，松开 */
13         input_sync(inputdev);                 /*  同步事件     */
14     }
15 }
```

第 6 行，获取按键值，判断按键是否按下。

第 9~10 行，如果按键值为 0 那么表示按键被按下了，如果按键按下的话就要使用 input_report_key 函数向 Linux 系统上报按键值，比如向 Linux 系统通知 KEY_0 这个按键按下了。

第 12~13 行，如果按键值为 1 的话就表示按键没有按下，是松开的。向 Linux 系统通知 KEY_0 这个按键没有按下或松开了。

58.1.3 input_event 结构体

Linux 内核使用 input_event 这个结构体来表示所有的输入事件，input_envent 结构体定义在 include/uapi/linux/input.h 文件中，结构体内容如下：

示例代码 58.1.3.1 input_event 结构体

```
24 struct input_event {
25     struct timeval time;
26     __u16 type;
27     __u16 code;
28     __s32 value;
29 };
```

我们依次来看一下 input_event 结构体中的各个成员变量：

time：时间，也就是此事件发生的时间，为 timeval 结构体类型，timeval 结构体定义如下：

示例代码 58.1.3.2 timeval 结构体

```
1 typedef long                __kernel_long_t;
2 typedef __kernel_long_t     __kernel_time_t;
3 typedef __kernel_long_t     __kernel_suseconds_t;
4 
5 struct timeval {
6       __kernel_time_t          tv_sec;    /* 秒   */
7       __kernel_suseconds_t     tv_usec;   /* 微秒     */
8 };
```

从示例代码 58.1.3.2 可以看出，tv_sec 和 tv_usec 这两个成员变量都为 long 类型，也就是 32 位，这个一定要记住，后面我们分析 event 事件上报数据的时候要用到。

type：事件类型，比如 EV_KEY，表示此次事件为按键事件，此成员变量为 16 位。

code：事件码，比如在 EV_KEY 事件中 code 就表示具体的按键码，如：KEY_0、KEY_1 等等这些按键。此成员变量为 16 位。

value：值，比如 EV_KEY 事件中 value 就是按键值，表示按键有没有被按下，如果为 1 的话说明按键按下，如果为 0 的话说明按键没有被按下或者按键松开了。

input_envent 这个结构体非常重要，因为所有的输入设备最终都是按照 input_event 结构体呈现给用户的，用户应用程序可以通过 input_event 来获取到具体的输入事件或相关的值，比如按键值等。关于 input 子系统就讲解到这里，接下来我们就以开发板上的 KEY0 按键为例，讲解一下如何编写 input 驱动。

58.2 硬件原理图分析

本章实验硬件原理图参考 15.2 小节即可。

58.3 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 20_input。

58.3.1 修改设备树文件

直接使用 49.3.1 小节创建的 key 节点即可。

58.3.2 按键 input 驱动程序编写

新建名为“20_input”的文件夹，然后在 20_input 文件夹里面创建 vscode 工程，工作区命名为“keyinput”。工程创建好以后新建 keyinput.c 文件，在 keyinput.c 里面输入如下内容：

示例代码 58.3.2.1 keyinput.c 文件代码段

```
1   #include <linux/types.h>
2   #include <linux/kernel.h>
3   #include <linux/delay.h>
4   #include <linux/ide.h>
5   #include <linux/init.h>
6   #include <linux/module.h>
```

```
7   #include <linux/errno.h>
8   #include <linux/gpio.h>
9   #include <linux/cdev.h>
10  #include <linux/device.h>
11  #include <linux/of.h>
12  #include <linux/of_address.h>
13  #include <linux/of_gpio.h>
14  #include <linux/input.h>
15  #include <linux/semaphore.h>
16  #include <linux/timer.h>
17  #include <linux/of_irq.h>
18  #include <linux/irq.h>
19  #include <asm/mach/map.h>
20  #include <asm/uaccess.h>
21  #include <asm/io.h>
22  /***************************************************************
23  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
24  文件名       : keyinput.c
25  作者         : 左忠凯
26  版本         : V1.0
27  描述         : Linux 按键 input 子系统实验
28  其他         : 无
29  论坛         : www.openedv.com
30  日志         : 初版 V1.0 2019/8/21 左忠凯创建
31  ***************************************************************/
32  #define KEYINPUT_CNT     1          /* 设备号个数   */
33  #define KEYINPUT_NAME    "keyinput" /* 名字         */
34  #define KEY0VALUE        0X01       /* KEY0 按键值  */
35  #define INVAKEY          0XFF       /* 无效的按键值 */
36  #define KEY_NUM          1          /* 按键数量     */
37
38  /* 中断 IO 描述结构体 */
39  struct irq_keydesc {
40      int gpio;                       /* gpio          */
41      int irqnum;                     /* 中断号        */
42      unsigned char value;            /* 按键对应的键值 */
43      char name[10];                  /* 名字          */
44      irqreturn_t (*handler)(int, void *);    /* 中断服务函数 */
45  };
46
47  /* keyinput 设备结构体 */
48  struct keyinput_dev{
49      dev_t devid;            /* 设备号          */
```

```
50      struct cdev cdev;         /* cdev             */
51      struct class *class;      /* 类               */
52      struct device *device;    /* 设备             */
53      struct device_node  *nd; /* 设备节点         */
54      struct timer_list timer; /* 定义一个定时器   */
55      struct irq_keydesc irqkeydesc[KEY_NUM];    /* 按键描述数组 */
56      unsigned char curkeynum;                   /* 当前的按键号  */
57      struct input_dev *inputdev;                /* input 结构体 */
58  };
59
60  struct keyinput_dev keyinputdev;    /* key input 设备 */
61
62  /* @description       : 中断服务函数，开启定时器，延时 10ms，
63   *                       定时器用于按键消抖。
64   * @param - irq        : 中断号
65   * @param - dev_id     : 设备结构。
66   * @return             : 中断执行结果
67   */
68  static irqreturn_t key0_handler(int irq, void *dev_id)
69  {
70      struct keyinput_dev *dev = (struct keyinput_dev *)dev_id;
71
72      dev->curkeynum = 0;
73      dev->timer.data = (volatile long)dev_id;
74      mod_timer(&dev->timer, jiffies + msecs_to_jiffies(10));
75      return IRQ_RETVAL(IRQ_HANDLED);
76  }
77
78  /* @description   : 定时器服务函数，用于按键消抖，定时器到了以后
79   *                  再次读取按键值，如果按键还是处于按下状态就表示按键有效。
80   * @param - arg    : 设备结构变量
81   * @return         : 无
82   */
83  void timer_function(unsigned long arg)
84  {
85      unsigned char value;
86      unsigned char num;
87      struct irq_keydesc *keydesc;
88      struct keyinput_dev *dev = (struct keyinput_dev *)arg;
89
90      num = dev->curkeynum;
91      keydesc = &dev->irqkeydesc[num];
92      value = gpio_get_value(keydesc->gpio);     /* 读取 IO 值    */
```

```
93      if(value == 0){                                     /* 按下按键     */
94          /* 上报按键值 */
95          //input_event(dev->inputdev, EV_KEY, keydesc->value, 1);
96          input_report_key(dev->inputdev, keydesc->value, 1);/*1，按下*/
97          input_sync(dev->inputdev);
98      } else {                                        /* 按键松开 */
99          //input_event(dev->inputdev, EV_KEY, keydesc->value, 0);
100         input_report_key(dev->inputdev, keydesc->value, 0);
101         input_sync(dev->inputdev);
102     }
103 }
104 
105 /*
106  * @description   : 按键 IO 初始化
107  * @param         : 无
108  * @return        : 无
109  */
110 static int keyio_init(void)
111 {
112     unsigned char i = 0;
113     char name[10];
114     int ret = 0;
115 
116     keyinputdev.nd = of_find_node_by_path("/key");
117     if (keyinputdev.nd== NULL){
118         printk("key node not find!\r\n");
119         return -EINVAL;
120     }
121 
122     /* 提取 GPIO */
123     for (i = 0; i < KEY_NUM; i++) {
124         keyinputdev.irqkeydesc[i].gpio =
                        of_get_named_gpio(keyinputdev.nd,"key-gpio", i);
125         if (keyinputdev.irqkeydesc[i].gpio < 0) {
126             printk("can't get key%d\r\n", i);
127         }
128     }
129 
130     /* 初始化 key 所使用的 IO，并且设置成中断模式 */
131     for (i = 0; i < KEY_NUM; i++) {
132         memset(keyinputdev.irqkeydesc[i].name, 0, sizeof(name));
133         sprintf(keyinputdev.irqkeydesc[i].name, "KEY%d", i);
134         gpio_request(keyinputdev.irqkeydesc[i].gpio, name);
```

```
135         gpio_direction_input(keyinputdev.irqkeydesc[i].gpio);
136         keyinputdev.irqkeydesc[i].irqnum = 
                                   irq_of_parse_and_map(keyinputdev.nd, i);
137     }
138     /* 申请中断 */
139     keyinputdev.irqkeydesc[0].handler = key0_handler;
140     keyinputdev.irqkeydesc[0].value = KEY_0;
141 
142     for (i = 0; i < KEY_NUM; i++) {
143         ret = request_irq(keyinputdev.irqkeydesc[i].irqnum,
                                 keyinputdev.irqkeydesc[i].handler,
144                            IRQF_TRIGGER_FALLING|IRQF_TRIGGER_RISING,
                             keyinputdev.irqkeydesc[i].name, &keyinputdev);
145         if(ret < 0){
146             printk("irq %d request failed!\r\n",
                        keyinputdev.irqkeydesc[i].irqnum);
147             return -EFAULT;
148         }
149     }
150 
151     /* 创建定时器 */
152     init_timer(&keyinputdev.timer);
153     keyinputdev.timer.function = timer_function;
154 
155     /* 申请 input_dev */
156     keyinputdev.inputdev = input_allocate_device();
157     keyinputdev.inputdev->name = KEYINPUT_NAME;
158 #if 0
159     /* 初始化 input_dev，设置产生哪些事件 */
160     __set_bit(EV_KEY, keyinputdev.inputdev->evbit); /*按键事件   */
161     __set_bit(EV_REP, keyinputdev.inputdev->evbit); /* 重复事件  */
162 
163     /* 初始化 input_dev，设置产生哪些按键 */
164     __set_bit(KEY_0, keyinputdev.inputdev->keybit);
165 #endif
166 
167 #if 0
168     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | 
                                         BIT_MASK(EV_REP);
169     keyinputdev.inputdev->keybit[BIT_WORD(KEY_0)] |= 
                                         BIT_MASK(KEY_0);
170 #endif
171 
```

```
172     keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) |
                                        BIT_MASK(EV_REP);
173     input_set_capability(keyinputdev.inputdev, EV_KEY, KEY_0);
174
175     /* 注册输入设备 */
176     ret = input_register_device(keyinputdev.inputdev);
177     if (ret) {
178         printk("register input device failed!\r\n");
179         return ret;
180     }
181     return 0;
182 }
183
184 /*
185  * @description   : 驱动入口函数
186  * @param         : 无
187  * @return        : 无
188  */
189 static int __init keyinput_init(void)
190 {
191     keyio_init();
192     return 0;
193 }
194
195 /*
196  * @description   : 驱动出口函数
197  * @param         : 无
198  * @return        : 无
199  */
200 static void __exit keyinput_exit(void)
201 {
202     unsigned i = 0;
203     /* 删除定时器 */
204     del_timer_sync(&keyinputdev.timer);
205
206     /* 释放中断 */
207     for (i = 0; i < KEY_NUM; i++) {
208         free_irq(keyinputdev.irqkeydesc[i].irqnum, &keyinputdev);
209     }
210     /* 释放 input_dev */
211     input_unregister_device(keyinputdev.inputdev);
212     input_free_device(keyinputdev.inputdev);
213 }
```

```
214
215 module_init(keyinput_init);
216 module_exit(keyinput_exit);
217 MODULE_LICENSE("GPL");
218 MODULE_AUTHOR("zuozhongkai");
```

keyinput.c 文件内容其实就是实验“13_irq”中的 imx6uirq.c 文件中修改而来的，只是将其中与字符设备有关的内容进行了删除，加入了 input_dev 相关的内容，我们简单来分析一下示例代码 58.3.2.1 中的程序。

第 57 行，在设备结构体中定义一个 input_dev 指针变量。

第 93~102 行，在按键消抖定时器处理函数中上报输入事件，也就是使用 input_report_key 函数上报按键事件以及按键值，最后使用 input_sync 函数上报一个同步事件，这一步一定得做！

第 156~180 行，使用 input_allocate_device 函数申请 input_dev，然后设置相应的事件以及事件码(也就是 KEY 模拟成那个按键，这里我们设置为 KEY_0)。最后使用 input_register_device 函数向 Linux 内核注册 input_dev。

第 211~212 行，当注销 input 设备驱动的时候使用 input_unregister_device 函数注销掉前面注册的 input_dev，最后使用 input_free_device 函数释放掉前面申请的 input_dev。

58.3.3 编写测试 APP

新建 keyinputApp.c 文件，然后在里面输入如下所示内容：

示例代码 58.3.3.1 keyinputApp.c 文件代码段

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "sys/ioctl.h"
6  #include "fcntl.h"
7  #include "stdlib.h"
8  #include "string.h"
9  #include <poll.h>
10 #include <sys/select.h>
11 #include <sys/time.h>
12 #include <signal.h>
13 #include <fcntl.h>
14 #include <linux/input.h>
15 /***************************************************************
16 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
17 文件名      : keyinputApp.c
18 作者        : 左忠凯
19 版本        : V1.0
20 描述        : input 子系统测试 APP。
21 其他        : 无
22 使用方法    ：./keyinputApp /dev/input/event1
23 论坛        : www.openedv.com
```

```
24 日志          : 初版 V1.0 2019/8/26 左忠凯创建
25 *****************************************************************/
26
27 /* 定义一个 input_event 变量，存放输入事件信息 */
28 static struct input_event inputevent;
29
30 /*
31  * @description    : main 主程序
32  * @param - argc   : argv 数组元素个数
33  * @param - argv   : 具体参数
34  * @return         : 0 成功;其他 失败
35  */
36 int main(int argc, char *argv[])
37 {
38     int fd;
39     int err = 0;
40     char *filename;
41
42     filename = argv[1];
43
44     if(argc != 2) {
45         printf("Error Usage!\r\n");
46         return -1;
47     }
48
49     fd = open(filename, O_RDWR);
50     if (fd < 0) {
51         printf("Can't open file %s\r\n", filename);
52         return -1;
53     }
54
55     while (1) {
56         err = read(fd, &inputevent, sizeof(inputevent));
57         if (err > 0) { /* 读取数据成功 */
58             switch (inputevent.type) {
59                 case EV_KEY:
60                     if (inputevent.code < BTN_MISC) { /* 键盘键值 */
61                         printf("key %d %s\r\n", inputevent.code,
                                inputevent.value ? "press" : "release");
62                     } else {
63                         printf("button %d %s\r\n", inputevent.code,
                                inputevent.value ? "press" : "release");
64                     }
```

```
65                     break;
66
67                 /* 其他类型的事件，自行处理 */
68                 case EV_REL:
69                     break;
70                 case EV_ABS:
71                     break;
72                 case EV_MSC:
73                     break;
74                 case EV_SW:
75                     break;
76             }
77         } else {
78             printf("读取数据失败\r\n");
79         }
80     }
81     return 0;
82 }
```

第 58.1.3 小节已经说过了，Linux 内核会使用 input_event 结构体来表示输入事件，所以我们要获取按键输入信息，那么必须借助于 input_event 结构体。第 28 行定义了一个 inputevent 变量，此变量为 input_event 结构体类型。

第 56 行，当我们向 Linux 内核成功注册 input_dev 设备以后，会在/dev/input 目录下生成一个名为“eventX(X=0….n)”的文件，这个/dev/input/eventX 就是对应的 input 设备文件。我们读取这个文件就可以获取到输入事件信息，比如按键值什么的。使用 read 函数读取输入设备文件，也就是/dev/input/eventX，读取到的数据按照 input_event 结构体组织起来。获取到输入事件以后(input_event 结构体类型)使用 switch case 语句来判断事件类型，本章实验我们设置的事件类型为 EV_KEY，因此只需要处理 EV_KEY 事件即可。比如获取按键编号(KEY_0 的编号为 11)、获取按键状态，按下还是松开的？

58.4 运行测试

58.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为“keyinput.o”，Makefile 内容如下所示：

示例代码 58.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
              rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := keyinput.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“keyinput.o”。
输入如下命令编译出驱动模块文件：
```
make -j32
```
编译成功以后就会生成一个名为“keyinput.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 keyinputApp.c 这个测试程序：
```
arm-linux-gnueabihf-gcc keyinputApp.c -o keyinputApp
```
编译成功以后就会生成 keyinputApp 这个应用程序。

58.4.2 运行测试

将上一小节编译出来 keyinput.ko 和 keyinputApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中。在加载 keyinput.ko 驱动模块之前，先看一下/dev/input 目录下都有哪些文件，结果如图 58.4.2.1 所示：

```
/ # ls /dev/input/ -l
total 0
crw-rw----    1 0        0          13,  64 Jan  1 00:00 event0
crw-rw----    1 0        0          13,  63 Jan  1 00:00 mice
/ #
```

图 58.4.2.1 当前/dev/input 目录文件

从图 58.4.2.1 可以看出，当前/dev/input 目录只有 event0 和 mice 这两个文件。接下来输入如下命令加载 keyinput.ko 这个驱动模块。
```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe keyinput.ko    //加载驱动模块
```
当驱动模块加载成功以后再来看一下/dev/input 目录下有哪些文件，结果如图 58.4.2.2 所示：

```
/lib/modules/4.1.15 # ls /dev/input/ -l
total 0
crw-rw----    1 0        0          13,  64 Jan  1 00:00 event0
crw-rw----    1 0        0          13,  65 Jan  1 00:41 event1
crw-rw----    1 0        0          13,  63 Jan  1 00:00 mice
/lib/modules/4.1.15 #
```

图 58.4.2.2 加载驱动以后的/dev/input 目录

从图 58.4.2.2 可以看出，多了一个 event1 文件，因此/dev/input/event1 就是我们注册的驱动所对应的设备文件。keyinputApp 就是通过读取/dev/input/event1 这个文件来获取输入事件信息的，输入如下测试命令：
```
./keyinputApp /dev/input/event1
```
然后按下开发板上的 KEY 按键，结果如图 58.4.2.3 所示：

```
/lib/modules/4.1.15 # ./keyinputApp /dev/input/event1
key 11 press
key 11 release
key 11 press
key 11 release
key 11 press
key 11 release
```

图 58.4.2.3 测试结果

从图 58.4.2.3 可以看出，当我们按下或者释放开发板上的按键以后都会在终端上输出相应的内容，提示我们哪个按键按下或释放了，在 Linux 内核中 KEY_0 为 11。

另外，我们也可以不用 keyinputApp 来测试驱动，可以直接使用 hexdump 命令来查看 /dev/input/event1 文件内容，输入如下命令：

```
hexdump /dev/input/event1
```

然后按下按键，终端输出如图 58.4.2.4 所示信息：

```
/lib/modules/4.1.15 # hexdump /dev/input/event1
0000000 0c41 0000 d7cd 000c 0001 000b 0001 0000
0000010 0c41 0000 d7cd 000c 0000 0000 0000 0000
0000020 0c42 0000 54bb 0000 0001 000b 0000 0000
0000030 0c42 0000 54bb 0000 0000 0000 0000 0000
0000040 0c42 0000 8909 0003 0001 000b 0001 0000
0000050 0c42 0000 8909 0003 0000 0000 0000 0000
0000060 0c42 0000 84ec 0005 0001 000b 0000 0000
0000070 0c42 0000 84ec 0005 0000 0000 0000 0000
0000080 0c42 0000 7c6c 0009 0001 000b 0001 0000
0000090 0c42 0000 7c6c 0009 0000 0000 0000 0000
00000a0 0c42 0000 0309 000b 0001 000b 0000 0000
00000b0 0c42 0000 0309 000b 0000 0000 0000 0000
```

图 58.4.2.4 原始数据值

图 58.4.2.4 就是 input_event 类型的原始事件数据值，采用十六进制表示，这些原始数据的含义如下：

示例代码 58.4.2.1 input_event 类型的原始事件值

```
                   /*****************input_event 类型********************/
/* 编号 */         /* tv_sec */  /* tv_usec */    /* type */   /* code */   /* value */
0000000            0c41 0000     d7cd 000c        0001         000b         0001 0000
0000010            0c41 0000     d7cd 000c        0000         0000         0000 0000
0000020            0c42 0000     54bb 0000        0001         000b         0000 0000
0000030            0c42 0000     54bb 0000        0000         0000         0000 0000
```

type 为事件类型，查看示例代码 58.1.2.3 可知，EV_KEY 事件值为 1，EV_SYN 事件值为 0。因此第 1 行表示 EV_KEY 事件，第 2 行表示 EV_SYN 事件。code 为事件编码，也就是按键号，查看示例代码 58.1.2.4 可以，KEY_0 这个按键编号为 11，对应的十六进制为 0xb，因此第 1 行表示 KEY_0 这个按键事件，最后的 value 就是按键值，为 1 表示按下，为 0 的话表示松开。综上所述，示例代码 58.4.2.1 中的原始事件值含义如下：

第 1 行，按键(KEY_0)按下事件。

第 2 行，EV_SYN 同步事件，因为每次上报按键事件以后都要同步的上报一个 EV_SYN 事件。

第 3 行，按键(KEY_0)松开事件。

第 4 行，EV_SYN 同步事件，和第 2 行一样。

58.5 Linux 自带按键驱动程序的使用

58.5.1 自带按键驱动程序源码简析

Linux 内核也自带了 KEY 驱动，如果要使用内核自带的 KEY 驱动的话需要配置 Linux 内

核，不过 Linux 内核一般默认已经使能了 KEY 驱动，但是我们还是要检查一下。按照如下路径找到相应的配置选项：

```
-> Device Drivers
    -> Input device support
        -> Generic input layer (needed for keyboard, mouse, ...) (INPUT [=y])
            -> Keyboards (INPUT_KEYBOARD [=y])
                ->GPIO Buttons
```

选中“GPIO Buttons”选项，将其编译进 Linux 内核中，如图 58.5.1.1 所示：

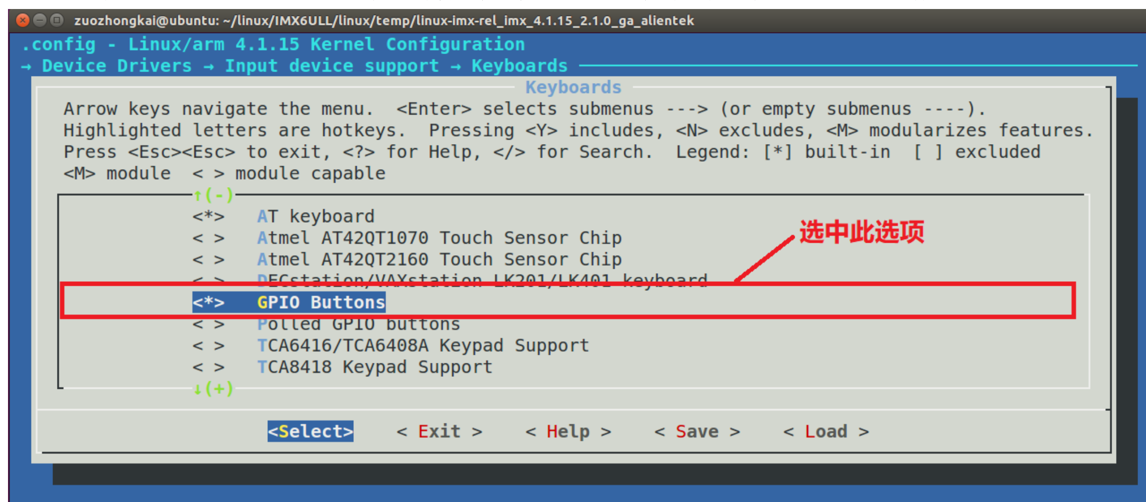


图 58.5.1.1 内核自带 KEY 驱动使能选项

选中以后就会在.config 文件中出现“CONFIG_KEYBOARD_GPIO=y”这一行，Linux 内核就会根据这一行来将 KEY 驱动文件编译进 Linux 内核。Linux 内核自带的 KEY 驱动文件为 drivers/input/keyboard/gpio_keys.c，gpio_keys.c 采用了 platform 驱动框架，在 KEY 驱动上使用了 input 子系统实现。在 gpio_keys.c 文件中找到如下所示内容：

示例代码 58.5.1.1 gpio_keys 文件代码段

```
673 static const struct of_device_id gpio_keys_of_match[] = {
674     { .compatible = "gpio-keys", },
675     { },
676 };
......
842 static struct platform_driver gpio_keys_device_driver = {
843     .probe      = gpio_keys_probe,
844     .remove     = gpio_keys_remove,
845     .driver     = {
846         .name   = "gpio-keys",
847         .pm = &gpio_keys_pm_ops,
848         .of_match_table = of_match_ptr(gpio_keys_of_match),
849     }
850 };
851 
852 static int __init gpio_keys_init(void)
853 {
```

```
854     return platform_driver_register(&gpio_keys_device_driver);
855 }
856 
857 static void __exit gpio_keys_exit(void)
858 {
859     platform_driver_unregister(&gpio_keys_device_driver);
860 }
```

从示例代码 58.5.1.1 可以看出，这就是一个标准的 platform 驱动框架，如果要使用设备树来描述 KEY 设备信息的话，设备节点的 compatible 属性值要设置为“gpio-keys”。当设备和驱动匹配以后 gpio_keys_probe 函数就会执行，gpio_keys_probe 函数内容如下(为了篇幅有缩减)：

示例代码 58.5.1.2 gpio_keys_probe 函数代码段

```
689 static int gpio_keys_probe(struct platform_device *pdev)
690 {
691     struct device *dev = &pdev->dev;
692     const struct gpio_keys_platform_data *pdata = 
                                dev_get_platdata(dev);
693     struct gpio_keys_drvdata *ddata;
694     struct input_dev *input;
695     size_t size;
696     int i, error;
697     int wakeup = 0;
698 
699     if (!pdata) {
700         pdata = gpio_keys_get_devtree_pdata(dev);
701         if (IS_ERR(pdata))
702             return PTR_ERR(pdata);
703     }
......
713     input = devm_input_allocate_device(dev);
714     if (!input) {
715         dev_err(dev, "failed to allocate input device\n");
716         return -ENOMEM;
717     }
718 
719     ddata->pdata = pdata;
720     ddata->input = input;
721     mutex_init(&ddata->disable_lock);
722 
723     platform_set_drvdata(pdev, ddata);
724     input_set_drvdata(input, ddata);
725 
726     input->name = pdata->name ? : pdev->name;
727     input->phys = "gpio-keys/input0";
```

```
728     input->dev.parent = &pdev->dev;
729     input->open = gpio_keys_open;
730     input->close = gpio_keys_close;
731 
732     input->id.bustype = BUS_HOST;
733     input->id.vendor = 0x0001;
734     input->id.product = 0x0001;
735     input->id.version = 0x0100;
736 
737     /* Enable auto repeat feature of Linux input subsystem */
738     if (pdata->rep)
739         __set_bit(EV_REP, input->evbit);
740 
741     for (i = 0; i < pdata->nbuttons; i++) {
742         const struct gpio_keys_button *button = &pdata->buttons[i];
743         struct gpio_button_data *bdata = &ddata->data[i];
744 
745         error = gpio_keys_setup_key(pdev, input, bdata, button);
746         if (error)
747             return error;
748 
749         if (button->wakeup)
750             wakeup = 1;
751     }
......
760     error = input_register_device(input);
761     if (error) {
762         dev_err(dev, "Unable to register input device, error: %d\n",
763             error);
764         goto err_remove_group;
765     }
......
774 }
```

第 700 行，调用 gpio_keys_get_devtree_pdata 函数从设备树中获取到 KEY 相关的设备节点信息。

第 713 行，使用 devm_input_allocate_device 函数申请 input_dev。

第 726~735，初始化 input_dev。

第 739 行，设置 input_dev 事件，这里设置了 EV_REP 事件。

第 745 行，调用 gpio_keys_setup_key 函数继续设置 KEY，此函数会设置 input_dev 的 EV_KEY 事件已经事件码(也就是 KEY 模拟为哪个按键)。

第 760 行，调用 input_register_device 函数向 Linux 系统注册 input_dev。

我们接下来再来看一下 gpio_keys_setup_key 函数，此函数内容如下：

示例代码 58.5.1.3 gpio_keys_setup_key 函数代码段

```
437 static int gpio_keys_setup_key(struct platform_device *pdev,
438                 struct input_dev *input,
439                 struct gpio_button_data *bdata,
440                 const struct gpio_keys_button *button)
441 {
442     const char *desc = button->desc ? button->desc : "gpio_keys";
443     struct device *dev = &pdev->dev;
444     irq_handler_t isr;
445     unsigned long irqflags;
446     int irq;
447     int error;
448 
449     bdata->input = input;
450     bdata->button = button;
451     spin_lock_init(&bdata->lock);
452 
453     if (gpio_is_valid(button->gpio)) {
454 
455         error = devm_gpio_request_one(&pdev->dev, button->gpio,
456                       GPIOF_IN, desc);
457         if (error < 0) {
458             dev_err(dev, "Failed to request GPIO %d, error %d\n",
459                 button->gpio, error);
460             return error;
......
488         isr = gpio_keys_gpio_isr;
489         irqflags = IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING;
490 
491     } else {
492         if (!button->irq) {
493             dev_err(dev, "No IRQ specified\n");
494             return -EINVAL;
495         }
496         bdata->irq = button->irq;
......
506 
507         isr = gpio_keys_irq_isr;
508         irqflags = 0;
509     }
510 
511     input_set_capability(input, button->type ?: EV_KEY,
                     button->code);
......
```

```
540     return 0;
541 }
```

第 511 行，调用 input_set_capability 函数设置 EV_KEY 事件以及 KEY 的按键类型，也就是 KEY 作为哪个按键？我们会在设备树里面设置指定的 KEY 作为哪个按键。

一切都准备就绪以后剩下的就是等待按键按下，然后向 Linux 内核上报事件，事件上报是在 gpio_keys_irq_isr 函数中完成的，此函数内容如下：

示例代码 58.5.1.4 gpio_keys_irq_isr 函数代码段

```
392 static irqreturn_t gpio_keys_irq_isr(int irq, void *dev_id)
393 {
394     struct gpio_button_data *bdata = dev_id;
395     const struct gpio_keys_button *button = bdata->button;
396     struct input_dev *input = bdata->input;
397     unsigned long flags;
398 
399     BUG_ON(irq != bdata->irq);
400 
401     spin_lock_irqsave(&bdata->lock, flags);
402 
403     if (!bdata->key_pressed) {
404         if (bdata->button->wakeup)
405             pm_wakeup_event(bdata->input->dev.parent, 0);
406 
407         input_event(input, EV_KEY, button->code, 1);
408         input_sync(input);
409 
410         if (!bdata->release_delay) {
411             input_event(input, EV_KEY, button->code, 0);
412             input_sync(input);
413             goto out;
414         }
415 
416         bdata->key_pressed = true;
417     }
418 
419     if (bdata->release_delay)
420         mod_timer(&bdata->release_timer,
421             jiffies + msecs_to_jiffies(bdata->release_delay));
422 out:
423     spin_unlock_irqrestore(&bdata->lock, flags);
424     return IRQ_HANDLED;
425 }
```

gpio_keys_irq_isr 是按键中断处理函数，第 407 行向 Linux 系统上报 EV_KEY 事件，表示按键按下。第 408 行使用 input_sync 函数向系统上报 EV_REP 同步事件。

综上所述，Linux 内核自带的 gpio_keys.c 驱动文件思路和我们前面编写的 keyinput.c 驱动文件基本一致。都是申请和初始化 input_dev，设置事件，向 Linux 内核注册 input_dev。最终在按键中断服务函数或者消抖定时器中断服务函数中上报事件和按键值。

58.5.2 自带按键驱动程序的使用

要使用 Linux 内核自带的按键驱动程序很简单，只需要根据 Documentation/devicetree/bindings/input/gpio-keys.txt 这个文件在设备树中添加指定的设备节点即可，节点要求如下：

①、节点名字为“gpio-keys”。

②、gpio-keys 节点的 compatible 属性值一定要设置为“gpio-keys”。

③、所有的 KEY 都是 gpio-keys 的子节点，每个子节点可以用如下属性描述自己：

gpios：KEY 所连接的 GPIO 信息。

interrupts：KEY 所使用 GPIO 中断信息，不是必须的，可以不写。

label：KEY 名字

linux,code：KEY 要模拟的按键，也就是示例代码 58.1.2.4 中的这些按键。

④、如果按键要支持连按的话要加入 autorepeat。

打开 imx6ull-alientek-emmc.dts，根据上面的要求创建对应的设备节点，设备节点内容如下所示：

示例代码 58.5.2.1 gpio-keys 节点内容

```
1  gpio-keys {
2      compatible = "gpio-keys";
3      #address-cells = <1>;
4      #size-cells = <0>;
5      autorepeat;
6      key0 {
7          label = "GPIO Key Enter";
8          linux,code = <KEY_ENTER>;
9          gpios = <&gpio1 18 GPIO_ACTIVE_LOW>;
10     };
11 };
```

第 5 行，autorepeat 表示按键支持连按。

第 6~10 行，ALPHA 开发板 KEY 按键信息，名字设置为“GPIO Key Enter”，这里我们将开发板上的 KEY 按键设置为“EKY_ENTER”这个按键，也就是回车键，效果和键盘上的回车键一样。后面学习 LCD 驱动的时候需要用到此按键，因为 Linux 内核设计的 10 分钟以后 LCD 关闭，也就是黑屏，就跟我们用电脑或者手机一样，一定时间以后关闭屏幕。这里将开发板上的 KEY 按键注册为回车键，当 LCD 黑屏以后直接按一下 KEY 按键即可唤醒屏幕，就跟当电脑熄屏以后按下回车键即可重新打开屏幕一样。

最后设置 KEY 所使用的 IO 为 GPIO1_IO18，一定要检查一下设备树看看此 GPIO 有没有被用到其他外设上，如果有的话要删除掉相关代码！

重新编译设备树，然后用新编译出来的 imx6ull-alientek-emmc.dtb 启动 Linux 系统，系统启动以后查看/dev/input 目录，看看都有哪些文件，结果如图 58.5.2.1 所示：

```
/ # ls /dev/input/ -l
total 0
crw-rw----    1 0        0          13,  64 Jan  1 00:00 event0
crw-rw----    1 0        0          13,  65 Jan  1 00:00 event1
crw-rw----    1 0        0          13.  63 Jan  1 00:00 mice
```

图 58.5.2.1 /dev/input 目录文件

从图 58.5.2.1 可以看出存在 event1 这个文件，这个文件就是 KEY 对应的设备文件，使用 hexdump 命令来查看/dev/input/event1 文件，输入如下命令：

```
hexdump /dev/input/event1
```

然后按下 ALPHA 开发板上的按键，终端输出图 58.5.2.2 所示内容：

```
/ # hexdump /dev/input/event1
0000000 0371 0000 a452 0002 0001 001c 0001 0000
0000010 0371 0000 a452 0002 0000 0000 0000 0000
0000020 0371 0000 8a8f 0005 0001 001c 0000 0000
0000030 0371 0000 8a8f 0005 0000 0000 0000 0000
0000040 0371 0000 4528 000a 0001 001c 0001 0000
0000050 0371 0000 4528 000a 0000 0000 0000 0000
0000060 0371 0000 6844 000c 0001 001c 0000 0000
0000070 0371 0000 6844 000c 0000 0000 0000 0000
```

图 58.5.2.2 按键信息

如果按下 KEY 按键以后会在终端上输出图 58.5.2.2 所示的信息那么就表示 Linux 内核的按键驱动工作正常。至于图 58.5.2.2 中内容的含义大家就自行分析，这个已经在 58.4.2 小节详细的分析过了，这里就不再讲解了。

大家如果发现按下 KEY 按键以后没有反应，那么请检查一下三方面：

①、是否使能 Linux 内核 KEY 驱动。

②、设备树中 gpio-keys 节点是否创建成功。

③、在设备树中是否有其他外设也使用了 KEY 按键对应的 GPIO，但是我们并没有删除掉这些外设信息。检查 Linux 启动 log 信息，看看是否有类似下面这条信息：

```
gpio-keys gpio_keys：Failed to request GPIO 18, error -16
```

上述信息表示 GPIO 18 申请失败，失败的原因就是有其他的外设正在使用此 GPIO。

第五十九章 Linux LCD 驱动实验

LCD 是很常用的一个外设，在裸机篇中我们讲解了如何编写 LCD 裸机驱动，在 Linux 下 LCD 的使用更加广泛，在搭配 QT 这样的 GUI 库下可以制作出非常精美的 UI 界面。本章我们就来学习一下如何在 Linux 下驱动 LCD 屏幕。

59.1 Linux 下 LCD 驱动简析

59.1.1 Framebuffer 设备

先来回顾一下裸机的时候 LCD 驱动是怎么编写的，裸机 LCD 驱动编写流程如下：

①、初始化 I.MX6U 的 eLCDIF 控制器，重点是 LCD 屏幕宽(width)、高(height)、hspw、hbp、hfp、vspw、vbp 和 vfp 等信息。

②、初始化 LCD 像素时钟。

③、设置 RGBLCD 显存。

④、应用程序直接通过操作显存来操作 LCD，实现在 LCD 上显示字符、图片等信息。

在 Linux 中应用程序最终也是通过操作 RGB LCD 的显存来实现在 LCD 上显示字符、图片等信息。在裸机中我们可以随意的分配显存，但是在 Linux 系统中内存的管理很严格，显存是不需要申请的，不是你想用就能用的。而且因为虚拟内存的存在，驱动程序设置的显存和应用程序访问的显存要是同一片物理内存。

为了解决上述问题，Framebuffer 诞生了， Framebuffer 翻译过来就是帧缓冲，简称 fb，因此大家在以后的 Linux 学习中见到"Framebuffer"或者"fb"的话第一反应应该想到 RGBLCD 或者显示设备。fb 是一种机制，将系统中所有跟显示有关的硬件以及软件集合起来，虚拟出一个 fb 设备，当我们编写好 LCD 驱动以后会生成一个名为/dev/fbX(X=0~n)的设备，应用程序通过访问/dev/fbX 这个设备就可以访问 LCD。NXP 官方的 Linux 内核默认已经开启了 LCD 驱动，因此我们是可以看到/dev/fb0 这样一个设备，如图 59.1.1.1 所示：

```
/ # ls /dev/fb* -l
crw-rw----    1 0        0         29,   0 Jan  1 00:00 /dev/fb0
/ #
```

图 59.1.1.1 /dev/fb0 设备文件

图 59.1.1.1 中的/dev/fb0 就是 LCD 对应的设备文件，/dev/fb0 是个字符设备，因此肯定有 file_operations 操作集，fb 的 file_operations 操作集定义在 drivers/video/fbdev/core/fbmem.c 文件中，如下所示：

示例代码 59.1.1.1 fb 设备的操作集

```
1495 static const struct file_operations fb_fops = {
1496    .owner     = THIS_MODULE,
1497    .read      = fb_read,
1498    .write     = fb_write,
1499    .unlocked_ioctl = fb_ioctl,
1500 #ifdef CONFIG_COMPAT
1501    .compat_ioctl = fb_compat_ioctl,
1502 #endif
1503    .mmap      = fb_mmap,
1504    .open      = fb_open,
1505    .release   = fb_release,
1506 #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
1507    .get_unmapped_area = get_fb_unmapped_area,
1508 #endif
1509 #ifdef CONFIG_FB_DEFERRED_IO
1510    .fsync     = fb_deferred_io_fsync,
```

```
1511 #endif
1512     .llseek    = default_llseek,
1513 };
```

关于 fb 的详细处理过程就不去深究了，本章我们的重点是驱动起来 ALPHA 开发板上的 LCD。

59.1.2 LCD 驱动简析

LCD 裸机例程主要分两部分：

①、获取 LCD 的屏幕参数。

②、根据屏幕参数信息来初始化 eLCDIF 接口控制器。

不同分辨率的 LCD 屏幕其 eLCDIF 控制器驱动代码都是一样的，只需要修改好对应的屏幕参数即可。屏幕参数信息属于屏幕设备信息内容，这些肯定是要放到设备树中的，因此我们本章实验的主要工作就是修改设备树，NXP 官方的设备树已经添加了 LCD 设备节点，只是此节点的 LCD 屏幕信息是针对 NXP 官方 EVK 开发板所使用的 4.3 寸 480*272 编写的，我们需要将其改为我们所使用的屏幕参数。

我们简单看一下 NXP 官方编写的 Linux 下的 LCD 驱动，打开 imx6ull.dtsi，然后找到 lcdif 节点内容，如下所示：

示例代码 59.1.2.1 imx6ull.dtsi 文件中 lcdif 节点内容

```
1  lcdif: lcdif@021c8000 {
2            compatible = "fsl,imx6ul-lcdif", "fsl,imx28-lcdif";
3            reg = <0x021c8000 0x4000>;
4            interrupts = <GIC_SPI 5 IRQ_TYPE_LEVEL_HIGH>;
5            clocks = <&clks IMX6UL_CLK_LCDIF_PIX>,
6                <&clks IMX6UL_CLK_LCDIF_APB>,
7                <&clks IMX6UL_CLK_DUMMY>;
8            clock-names = "pix", "axi", "disp_axi";
9            status = "disabled";
10 };
```

示例代码 59.1.2.1 中的 lcdif 节点信息是所有使用 I.MX6ULL 芯片的板子所共有的，并不是完整的 lcdif 节点信息。像屏幕参数这些需要根据不同的硬件平台去添加，比如 imx6ull-alientek-emmc.dts 文件中会想 lcdif 节点添加其他的属性信息。从示例代码 59.1.2.1 可以看出 lcdif 节点的 compatible 属性值为“fsl,imx6ul-lcdif”和“fsl,imx28-lcdi”，因此在 Linux 源码中搜索这两个字符串即可找到 I.MX6ULL 的 LCD 驱动文件，这个文件为 drivers/video/fbdev/mxsfb.c，mxsfb.c 就是 I.MX6ULL 的 LCD 驱动文件，在此文件中找到如下内容：

示例代码 59.1.2.2 platform 下的 LCD 驱动

```
1362 static const struct of_device_id mxsfb_dt_ids[] = {
1363     { .compatible = "fsl,imx23-lcdif", .data = &mxsfb_devtype[0], },
1364     { .compatible = "fsl,imx28-lcdif", .data = &mxsfb_devtype[1], },
1365     { /* sentinel */ }
1366 };
......
1625 static struct platform_driver mxsfb_driver = {
1626     .probe = mxsfb_probe,
```

```
1627    .remove = mxsfb_remove,
1628    .shutdown = mxsfb_shutdown,
1629    .id_table = mxsfb_devtype,
1630    .driver = {
1631            .name = DRIVER_NAME,
1632            .of_match_table = mxsfb_dt_ids,
1633            .pm = &mxsfb_pm_ops,
1634    },
1635 };
1636
1637 module_platform_driver(mxsfb_driver);
```

从示例代码 59.1.2.2 可以看出，这是一个标准的 platform 驱动，当驱动和设备匹配以后 mxsfb_probe 函数就会执行。在看 mxsfb_probe 函数之前我们先简单了解一下 Linux 下 Framebuffer 驱动的编写流程，Linux 内核将所有的 Framebuffer 抽象为一个叫做 fb_info 的结构体，fb_info 结构体包含了 Framebuffer 设备的完整属性和操作集合，因此每一个 Framebuffer 设备都必须有一个 fb_info。换言之就是，LCD 的驱动就是构建 fb_info，并且向系统注册 fb_info 的过程。fb_info 结构体定义在 include/linux/fb.h 文件里面，内容如下(省略掉条件编译)：

示例代码 59.1.2.3 fb_info 结构体

```
448 struct fb_info {
449     atomic_t count;
450     int node;
451     int flags;
452     struct mutex lock;              /* 互斥锁           */
453     struct mutex mm_lock;           /* 互斥锁，用于 fb_mmap 和 smem_*域*/
454     struct fb_var_screeninfo var;  /* 当前可变参数      */
455     struct fb_fix_screeninfo fix;  /* 当前固定参数      */
456     struct fb_monspecs monspecs;   /* 当前显示器特性    */
457     struct work_struct queue;      /* 帧缓冲事件队列    */
458     struct fb_pixmap pixmap;       /* 图像硬件映射      */
459     struct fb_pixmap sprite;       /* 光标硬件映射      */
460     struct fb_cmap cmap;           /* 当前调色板        */
461     struct list_head modelist;     /* 当前模式列表      */
462     struct fb_videomode *mode;     /* 当前视频模式      */
463
464 #ifdef CONFIG_FB_BACKLIGHT          /* 如果 LCD 支持背光的话 */
465     /* assigned backlight device */
466     /* set before framebuffer registration,
467        remove after unregister */
468     struct backlight_device *bl_dev;   /* 背光设备      */
469
470     /* Backlight level curve */
471     struct mutex bl_curve_mutex;
472     u8 bl_curve[FB_BACKLIGHT_LEVELS];
```

```
473 #endif
......
479     struct fb_ops *fbops;           /* 帧缓冲操作函数集  */
480     struct device *device;          /* 父设备            */
481     struct device *dev;             /* 当前 fb 设备      */
482     int class_flag;                 /* 私有 sysfs 标志   */
......
486     char __iomem *screen_base;      /* 虚拟内存基地址(屏幕显存)      */
487     unsigned long screen_size;      /* 虚拟内存大小(屏幕显存大小)    */
488     void *pseudo_palette;           /* 伪 16 位调色板                */
......
507 };
```

fb_info 结构体的成员变量很多，我们重点关注 var、fix、fbops、screen_base、screen_size 和 pseudo_palette。mxsfb_probe 函数的主要工作内容为：

①、申请 fb_info。

②、初始化 fb_info 结构体中的各个成员变量。

③、初始化 eLCDIF 控制器。

④、使用 register_framebuffer 函数向 Linux 内核注册初始化好的 fb_info。register_framebuffer 函数原型如下：

```
int register_framebuffer(struct fb_info *fb_info)
```

函数参数和返回值含义如下：

fb_info：需要上报的 fb_info。

返回值：0，成功；负值，失败。

接下来我们简单看一下 mxsfb_probe 函数，函数内容如下(有缩减)：

示例代码 59.1.2.4 mxsfb_probe 函数

```
1369 static int mxsfb_probe(struct platform_device *pdev)
1370 {
1371     const struct of_device_id *of_id =
1372             of_match_device(mxsfb_dt_ids, &pdev->dev);
1373     struct resource *res;
1374     struct mxsfb_info *host;
1375     struct fb_info *fb_info;
1376     struct pinctrl *pinctrl;
1377     int irq = platform_get_irq(pdev, 0);
1378     int gpio, ret;
1379 
......
1394 
1395     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
1396     if (!res) {
1397         dev_err(&pdev->dev, "Cannot get memory IO resource\n");
1398         return -ENODEV;
1399     }
```

```
1400
1401    host = devm_kzalloc(&pdev->dev, sizeof(struct mxsfb_info),
                                  GFP_KERNEL);
1402    if (!host) {
1403        dev_err(&pdev->dev, "Failed to allocate IO resource\n");
1404        return -ENOMEM;
1405    }
1406
1407    fb_info = framebuffer_alloc(sizeof(struct fb_info), &pdev->dev);
1408    if (!fb_info) {
1409        dev_err(&pdev->dev, "Failed to allocate fbdev\n");
1410        devm_kfree(&pdev->dev, host);
1411        return -ENOMEM;
1412    }
1413    host->fb_info = fb_info;
1414    fb_info->par = host;
1415
1416    ret = devm_request_irq(&pdev->dev, irq, mxsfb_irq_handler, 0,
1417              dev_name(&pdev->dev), host);
1418    if (ret) {
1419        dev_err(&pdev->dev, "request_irq (%d) failed with
1420                error %d\n", irq, ret);
1421        ret = -ENODEV;
1422        goto fb_release;
1423    }
1424
1425    host->base = devm_ioremap_resource(&pdev->dev, res);
1426    if (IS_ERR(host->base)) {
1427        dev_err(&pdev->dev, "ioremap failed\n");
1428        ret = PTR_ERR(host->base);
1429        goto fb_release;
1430    }
......
1461
1462    fb_info->pseudo_palette = devm_kzalloc(&pdev->dev, sizeof(u32) *
1463                                          16, GFP_KERNEL);
1464    if (!fb_info->pseudo_palette) {
1465        ret = -ENOMEM;
1466        goto fb_release;
1467    }
1468
1469    INIT_LIST_HEAD(&fb_info->modelist);
1470
```

```
1471    pm_runtime_enable(&host->pdev->dev);
1472
1473    ret = mxsfb_init_fbinfo(host);
1474    if (ret != 0)
1475        goto fb_pm_runtime_disable;
1476
1477    mxsfb_dispdrv_init(pdev, fb_info);
1478
1479    if (!host->dispdrv) {
1480        pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
1481        if (IS_ERR(pinctrl)) {
1482            ret = PTR_ERR(pinctrl);
1483            goto fb_pm_runtime_disable;
1484        }
1485    }
1486
1487    if (!host->enabled) {
1488        writel(0, host->base + LCDC_CTRL);
1489        mxsfb_set_par(fb_info);
1490        mxsfb_enable_controller(fb_info);
1491        pm_runtime_get_sync(&host->pdev->dev);
1492    }
1493
1494    ret = register_framebuffer(fb_info);
1495    if (ret != 0) {
1496        dev_err(&pdev->dev, "Failed to register framebuffer\n");
1497        goto fb_destroy;
1498    }
......
1525    return ret;
1526 }
```

第 1374 行，host 结构体指针变量，表示 I.MX6ULL 的 LCD 的主控接口，mxsfb_info 结构体是 NXP 定义的针对 I.MX 系列 SOC 的 Framebuffer 设备结构体。也就是我们前面一直说的设备结构体，此结构体包含了 I.MX 系列 SOC 的 Framebuffer 设备详细信息，比如时钟、eLCDIF 控制器寄存器基地址、fb_info 等。

第 1395 行，从设备树中获取 eLCDIF 接口控制器的寄存器首地址，设备树中 lcdif 节点已经设置了 eLCDIF 寄存器首地址为 0X021C8000，因此 res=0X021C8000。

第 1401 行，给 host 申请内存，host 为 mxsfb_info 类型结构体指针。

第 1407 行，给 fb_info 申请内存，也就是申请 fb_info。

第 1413~1414 行，设置 host 的 fb_info 成员变量为 fb_info，设置 fb_info 的 par 成员变量为 host。通过这一步就将前面申请的 host 和 fb_info 联系在了一起。

第 1416 行，申请中断，中断服务函数为 mxsfb_irq_handler。

第 1425 行，对从设备树中获取到的寄存器首地址(res)进行内存映射，得到虚拟地址，并保存到 host 的 base 成员变量。因此通过访问 host 的 base 成员即可访问 I.MX6ULL 的整个 eLCDIF 寄存器。其实在 mxsfb.c 中已经定义了 eLCDIF 各个寄存器相比于基地址的偏移值，如下所示：

示例代码 59.1.2.4 eLCDIF 各个寄存器偏移值

```
67 #define LCDC_CTRL                      0x00
68 #define LCDC_CTRL1                     0x10
69 #define LCDC_V4_CTRL2                  0x20
70 #define LCDC_V3_TRANSFER_COUNT         0x20
71 #define LCDC_V4_TRANSFER_COUNT         0x30
......
89 #define LCDC_V4_DEBUG0                 0x1d0
90 #define LCDC_V3_DEBUG0                 0x1f0
```

大家可以对比着《I.MX6ULL 参考手册》中的 eLCDIF 章节检查一下示例代码 59.1.2.4 中的这些寄存器有没有错误。

继续回到示例代码 59.1.2.5 中的 mxsfb_probe 函数，第 1462 行，给 fb_info 中的 pseudo_palette 申请内存。

第 1473 行，调用 mxsfb_init_fbinfo 函数初始化 fb_info，重点是 fb_info 的 var、fix、fbops，screen_base 和 screen_size。其中 fbops 是 Framebuffer 设备的操作集，NXP 提供的 fbops 为 mxsfb_ops，内容如下：

示例代码 59.1.2.5 mxsfb_ops 操作集合

```
987 static struct fb_ops mxsfb_ops = {
988     .owner = THIS_MODULE,
989     .fb_check_var = mxsfb_check_var,
990     .fb_set_par = mxsfb_set_par,
991     .fb_setcolreg = mxsfb_setcolreg,
992     .fb_ioctl = mxsfb_ioctl,
993     .fb_blank = mxsfb_blank,
994     .fb_pan_display = mxsfb_pan_display,
995     .fb_mmap = mxsfb_mmap,
996     .fb_fillrect = cfb_fillrect,
997     .fb_copyarea = cfb_copyarea,
998     .fb_imageblit = cfb_imageblit,
999 };
```

关于 mxsfb_ops 里面的各个操作函数这里就不去详解的介绍了。mxsfb_init_fbinfo 函数通过调用 mxsfb_init_fbinfo_dt 函数从设备树中获取到 LCD 的各个参数信息。最后，mxsfb_init_fbinfo 函数会调用 mxsfb_map_videomem 函数申请 LCD 的帧缓冲内存(也就是显存)。

第 1489~1490 行，设置 eLCDIF 控制器的相应寄存器。

第 1494 行，最后调用 register_framebuffer 函数向 Linux 内核注册 fb_info。

mxsfb.c 文件很大，还有一些其他的重要函数，比如 mxsfb_remove、mxsfb_shutdown 等，这里我们就简单的介绍了一下 mxsfb_probe 函数，至于其他的函数大家自行查阅。

59.2 硬件原理图分析

本章实验硬件原理图参考 24.2 小节即可。

59.3 LCD 驱动程序编写

前面已经说了，6ULL 的 eLCDIF 接口驱动程序 NXP 已经编写好了，因此 LCD 驱动部分我们不需要去修改。我们需要做的就是按照所使用的 LCD 来修改设备树。重点要注意三个地方：

①、LCD 所使用的 IO 配置。

①、LCD 屏幕节点修改，修改相应的属性值，换成我们所使用的 LCD 屏幕参数。

②、LCD 背光节点信息修改，要根据实际所使用的背光 IO 来修改相应的设备节点信息。

接下来我们依次来看一下上面这两个节点改如何去修改：

1、LCD 屏幕 IO 配置

首先要检查一下设备树中 LCD 所使用的 IO 配置，这个其实 NXP 都已经给我们写好了，不需要修改，不过我们还是要看一下。打开 imx6ull-alientek-emmc.dts 文件，在 iomuxc 节点中找到如下内容：

示例代码 59.3.1 设备树 LCD IO 配置

```
1  pinctrl_lcdif_dat: lcdifdatgrp {
2      fsl,pins = <
3          MX6UL_PAD_LCD_DATA00__LCDIF_DATA00  0x79
4          MX6UL_PAD_LCD_DATA01__LCDIF_DATA01  0x79
5          MX6UL_PAD_LCD_DATA02__LCDIF_DATA02  0x79
6          MX6UL_PAD_LCD_DATA03__LCDIF_DATA03  0x79
7          MX6UL_PAD_LCD_DATA04__LCDIF_DATA04  0x79
8          MX6UL_PAD_LCD_DATA05__LCDIF_DATA05  0x79
9          MX6UL_PAD_LCD_DATA06__LCDIF_DATA06  0x79
10         MX6UL_PAD_LCD_DATA07__LCDIF_DATA07  0x79
11         MX6UL_PAD_LCD_DATA08__LCDIF_DATA08  0x79
12         MX6UL_PAD_LCD_DATA09__LCDIF_DATA09  0x79
13         MX6UL_PAD_LCD_DATA10__LCDIF_DATA10  0x79
14         MX6UL_PAD_LCD_DATA11__LCDIF_DATA11  0x79
15         MX6UL_PAD_LCD_DATA12__LCDIF_DATA12  0x79
16         MX6UL_PAD_LCD_DATA13__LCDIF_DATA13  0x79
17         MX6UL_PAD_LCD_DATA14__LCDIF_DATA14  0x79
18         MX6UL_PAD_LCD_DATA15__LCDIF_DATA15  0x79
19         MX6UL_PAD_LCD_DATA16__LCDIF_DATA16  0x79
20         MX6UL_PAD_LCD_DATA17__LCDIF_DATA17  0x79
21         MX6UL_PAD_LCD_DATA18__LCDIF_DATA18  0x79
22         MX6UL_PAD_LCD_DATA19__LCDIF_DATA19  0x79
23         MX6UL_PAD_LCD_DATA20__LCDIF_DATA20  0x79
24         MX6UL_PAD_LCD_DATA21__LCDIF_DATA21  0x79
25         MX6UL_PAD_LCD_DATA22__LCDIF_DATA22  0x79
26         MX6UL_PAD_LCD_DATA23__LCDIF_DATA23  0x79
27     >;
28 };
29
```

```
30 pinctrl_lcdif_ctrl: lcdifctrlgrp {
31     fsl,pins = <
32         MX6UL_PAD_LCD_CLK__LCDIF_CLK          0x79
33         MX6UL_PAD_LCD_ENABLE__LCDIF_ENABLE    0x79
34         MX6UL_PAD_LCD_HSYNC__LCDIF_HSYNC      0x79
35         MX6UL_PAD_LCD_VSYNC__LCDIF_VSYNC      0x79
36     >;
37 pinctrl_pwm1: pwm1grp {
38     fsl,pins = <
39         MX6UL_PAD_GPIO1_IO08__PWM1_OUT        0x110b0
40     >;
41 };
```

第 2~27 行，子节点 pinctrl_lcdif_dat，为 RGB LCD 的 24 根数据线配置项。

第 30~36 行，子节点 pinctrl_lcdif_ctrl，RGB LCD 的 4 根控制线配置项，包括 CLK、ENABLE、VSYNC 和 HSYNC。

第 37~40 行，子节点 pinctrl_pwm1，LCD 背光 PWM 引脚配置项。这个引脚要根据实际情况设置，这里我们建议大家在以后的学习或工作中，LCD 的背光 IO 尽量和半导体厂商的官方开发板一致。

2、LCD 屏幕参数节点信息修改

继续在 imx6ull-alientek-emmc.dts 文件中找到 lcdif 节点，节点内容如下所示：

示例代码 59.3.2 lcdif 节点默认信息

```
1  &lcdif {
2      pinctrl-names = "default";
3      pinctrl-0 = <&pinctrl_lcdif_dat          /* 使用到的 IO      */
4              &pinctrl_lcdif_ctrl
5              &pinctrl_lcdif_reset>;
6      display = <&display0>;
7      status = "okay";
8  
9      display0: display {                    /* LCD 属性信息         */
10         bits-per-pixel = <16>;             /* 一个像素占用几个 bit  */
11         bus-width = <24>;                  /* 总线宽度             */
12 
13         display-timings {
14             native-mode = <&timing0>;      /* 时序信息             */
15             timing0: timing0 {
16             clock-frequency = <9200000>; /* LCD 像素时钟，单位 Hz */
17             hactive = <480>;               /* LCD X 轴像素个数      */
18             vactive = <272>;               /* LCD Y 轴像素个数      */
19             hfront-porch = <8>;            /* LCD hfp 参数          */
20             hback-porch = <4>;             /* LCD hbp 参数          */
21             hsync-len = <41>;              /* LCD hspw 参数         */
22             vback-porch = <2>;             /* LCD vbp 参数          */
```

```
23              vfront-porch = <4>;            /* LCD vfp 参数         */
24              vsync-len = <10>;              /* LCD vspw 参数        */
25
26              hsync-active = <0>;            /* hsync 数据线极性     */
27              vsync-active = <0>;            /* vsync 数据线极性     */
28              de-active = <1>;               /* de 数据线极性        */
29              pixelclk-active = <0>;         /* clk 数据线先极性     */
30          };
31      };
32   };
33 };
```

示例代码 59.3.2 就是向 imx6ull.dtsi 文件中的 lcdif 节点追加的内容，我们依次来看一下示例代码 59.3.2 中的这些属性都是写什么含义。

第 3 行，pinctrl-0 属性，LCD 所使用的 IO 信息，这里用到了 pinctrl_lcdif_dat、pinctrl_lcdif_ctrl 和 pinctrl_lcdif_reset 这三个 IO 相关的节点，前两个在示例代码 59.3.1 中已经讲解了。pinctrl_lcdif_reset 是 LCD 复位 IO 信息节点，正点原子的 I.MX6U-ALPHA 开发板的 LCD 没有用到复位 IO，因此 pinctrl_lcdif_reset 可以删除掉。

第 6 行，display 属性，指定 LCD 属性信息所在的子节点，这里为 display0，下面就是 display0 子节点内容。

第 9~32 行，display0 子节点，描述 LCD 的参数信息，第 10 行的 bits-per-pixel 属性用于指明一个像素占用的 bit 数，默认为 16bit。本教程我们将 LCD 配置为 RGB888 模式，因此一个像素点占用 24bit，bits-per-pixel 属性要改为 24。第 11 行的 bus-width 属性用于设置数据线宽度，因为要配置为 RGB888 模式，因此 bus-width 也要设置为 24。

第 13~30 行，这几行非常重要！因为这几行设置了 LCD 的时序参数信息，NXP 官方的 EVK 开发板使用了一个 4.3 寸的 480*272 屏幕，因此这里默认是按照 NXP 官方的那个屏幕参数设置的。每一个属性的含义后面的注释已经写的很详细了，大家自己去看就行了，这些时序参数就是我们重点要修改的，需要根据自己所使用的屏幕去修改。

这里以正点原子的 ATK7016(7 寸 1024*600)屏幕为例，将 imx6ull-alientek-emmc.dts 文件中的 lcdif 节点改为如下内容：

示例代码 59.3.3 针对 ATK7016 LCD 修改后的 lcdif 节点信息

```
1  &lcdif {
2      pinctrl-names = "default";
3      pinctrl-0 = <&pinctrl_lcdif_dat       /* 使用到的 IO          */
4                   &pinctrl_lcdif_ctrl>;
5      display = <&display0>;
6      status = "okay";
7
8      display0: display {                    /* LCD 属性信息         */
9          bits-per-pixel = <24>;             /* 一个像素占用 24bit   */
10         bus-width = <24>;                  /* 总线宽度             */
11
12         display-timings {
13             native-mode = <&timing0>;      /* 时序信息             */
```

```
14              timing0: timing0 {
15              clock-frequency = <51200000>;/* LCD 像素时钟，单位 Hz */
16              hactive = <1024>;           /* LCD X 轴像素个数    */
17              vactive = <600>;            /* LCD Y 轴像素个数    */
18              hfront-porch = <160>;       /* LCD hfp 参数        */
19              hback-porch = <140>;        /* LCD hbp 参数        */
20              hsync-len = <20>;           /* LCD hspw 参数       */
21              vback-porch = <20>;         /* LCD vbp 参数        */
22              vfront-porch = <12>;        /* LCD vfp 参数        */
23              vsync-len = <3>;            /* LCD vspw 参数       */
24
25              hsync-active = <0>;         /* hsync 数据线极性    */
26              vsync-active = <0>;         /* vsync 数据线极性    */
27              de-active = <1>;            /* de 数据线极性       */
28              pixelclk-active = <0>;      /* clk 数据线先极性    */
29              };
30          };
31      };
32  };
```

第 3 行，设置 LCD 屏幕所使用的 IO，删除掉原来的 pinctrl_lcdif_reset，因为没有用到屏幕复位 IO，其他的 IO 不变。

第 9 行，使用 RGB888 模式，所以一个像素点是 24bit。

第 15~23 行，ATK7016 屏幕时序参数，根据自己所使用的屏幕修改即可。

3、LCD 屏幕背光节点信息

正点原子的 LCD 接口背光控制 IO 连接到了 I.MX6U 的 GPIO1_IO08 引脚上，GPIO1_IO08 复用为 PWM1_OUT，通过 PWM 信号来控制 LCD 屏幕背光的亮度，这个我们已经在第二十九章详细的讲解过了。正点原子 I.MX6U-ALPHA 开发板的 LCD 背光引脚和 NXP 官方 EVK 开发板的背光引脚一样，因此背光的设备树节点是不需要修改的，但是考虑到其他同学可能使用别的开发板或者屏幕，LCD 背光引脚和 NXP 官方 EVK 开发板可能不同，因此我们还是来看一下如何在设备树中添加背光节点信息。

首先是 GPIO1_IO08 这个 IO 的配置，在 imx6ull-alientek-emmc.dts 中找到如下内容：

示例代码 59.3.4 GPIO1_IO08 引脚配置

```
1 pinctrl_pwm1: pwm1grp {
2     fsl,pins = <
3         MX6UL_PAD_GPIO1_IO08__PWM1_OUT    0x110b0
4     >;
5 };
```

pinctrl_pwm1 节点就是 GPIO1_IO08 的配置节点，从第 3 行可以看出，设置 GPIO1_IO08 这个 IO 复用为 PWM1_OUT，并且设置电气属性值为 0x110b0。

LCD 背光要用到 PWM1，因此也要设置 PWM1 节点，在 imx6ull.dtsi 文件中找到如下内容：

示例代码 59.3.5 imx6ull.dtsi 文件中的 pwm1 节点

```
1 pwm1: pwm@02080000 {
```

```
2     compatible = "fsl,imx6ul-pwm", "fsl,imx27-pwm";
3     reg = <0x02080000 0x4000>;
4     interrupts = <GIC_SPI 83 IRQ_TYPE_LEVEL_HIGH>;
5     clocks = <&clks IMX6UL_CLK_PWM1>,
6         <&clks IMX6UL_CLK_PWM1>;
7     clock-names = "ipg", "per";
8     #pwm-cells = <2>;
9 };
```

imx6ull.dtsi 文件中的 pwm1 节点信息大家不要修改，如果要修改 pwm1 节点内容的话请在 imx6ull-alientek-emmc.dts 文件中修改。在整个 Linux 源码文件中搜索 compatible 属性的这两个值即可找到 imx6ull 的 pwm 驱动文件，imx6ull 的 PWM 驱动文件为 drivers/pwm/pwm-imx.c，这里我们就不详细的去分析这个文件了。继续在 imx6ull-alientek-emmc.dts 文件中找到向 pwm1 追加的内容，如下所示：

示例代码 59.3.6 向 pwm1 节点追加的内容

```
1 &pwm1 {
2     pinctrl-names = "default";
3     pinctrl-0 = <&pinctrl_pwm1>;
4     status = "okay";
5 };
```

第 3 行，设置 pwm1 所使用的 IO 为 pinctrl_pwm1，也就是示例代码 59.3.4 所定义的 GPIO1_IO08 这个 IO。

第 4 行，将 status 设置为 okay。

如果背光用的路其他 pwm，比如 pwm2，那么就需要仿照示例代码 59.3.6 的内容，向 pwm2 节点追加相应的内容。pwm 和相关的 IO 已经准备好了，但是 Linux 系统怎么知道 PWM1_OUT 就是控制 LCD 背光的呢？因此我们还需要一个节点来将 LCD 背光和 PWM1_OUT 连接起来。这个节点就是 backlight，backlight 节点描述可以参考 Documentation/devicetree/indings/video/backlight/pwm-backlight.txt 这个文档，此文档详细讲解了 backlight 节点该如何去创建，这里大概总结一下：

①、节点名称要为“backlight”。

②、节点的 compatible 属性值要为“pwm-backlight”，因此可以通过在 Linux 内核中搜索“pwm-backlight”来查找 PWM 背光控制驱动程序，这个驱动程序文件为 drivers/video/backlight/pwm_bl.c，感兴趣的可以去看一下这个驱动程序。

③、pwms 属性用于描述背光所使用的 PWM 以及 PWM 频率，比如本章我们要使用的 pwm1，pwm 频率设置为 5KHz(NXP 官方推荐设置)。

④、brightness-levels 属性描述亮度级别，范围为 0~255，0 表示 PWM 占空比为 0%，也就是亮度最低，255 表示 100%占空比，也就是亮度最高。至于设置几级亮度，大家可以自行填写此属性。

⑤、default-brightness-level 属性为默认亮度级别。

根据上述 5 点设置 backlight 节点，这个 NXP 已经给我们设置好了，大家在 imx6ull-alientek-emmc.dts 文件中找到如下内容：

示例代码 59.3.7 backlight 节点内容

```
1 backlight {
2     compatible = "pwm-backlight";
```

```
3     pwms = <&pwm1 0 5000000>;
4     brightness-levels = <0 4 8 16 32 64 128 255>;
5     default-brightness-level = <6>;
6     status = "okay";
7 };
```

第 3 行，设置背光使用 pwm1，PWM 频率为 5KHz。

第 4 行，设置背 8 级背光(0~7)，分别为 0、4、8、16、32、64、128、255，对应占空比为 0%、1.57%、3.13%、6.27%、12.55%、25.1%、50.19%、100%，如果嫌少的话可以自行添加一些其他的背光等级值。

第 5 行，设置默认背光等级为 6，也就是 50.19%的亮度。

关于背光的设备树节点信息就讲到这里，整个的 LCD 设备树节点内容我们就讲完了，按照这些节点内容配置自己的开发板即可。

59.4 运行测试

59.4.1 LCD 屏幕基本测试

1、编译新的设备树

上一小节我们已经配置好了设备树，所以需要输入如下命令重新编译一下设备树：

```
make dtbs
```

等待编译生成新的 imx6ull-alientek-emmc.dtb 设备树文件，一会要使用新的设备树启动 Linux 内核。

2、使能 Linux logo 显示

Linux 内核启动的时候可以选择显示小企鹅 logo，只要这个小企鹅 logo 显示没问题那么我们的 LCD 驱动基本就工作正常了。这个 logo 显示是要配置的，不过 Linux 内核一般都会默认开启 logo 显示，但是奔着学习的目的，我们还是来看一下如何使能 Linux logo 显示。打开 Linux 内核图形化配置界面，按下路径找到对应的配置项：

```
-> Device Drivers
    -> Graphics support
        -> Bootup logo (LOGO [=y])
            -> Standard black and white Linux logo
            -> Standard 16-color Linux logo
            -> Standard 224-color Linux logo
```

如图 59.4.1.1 所示：

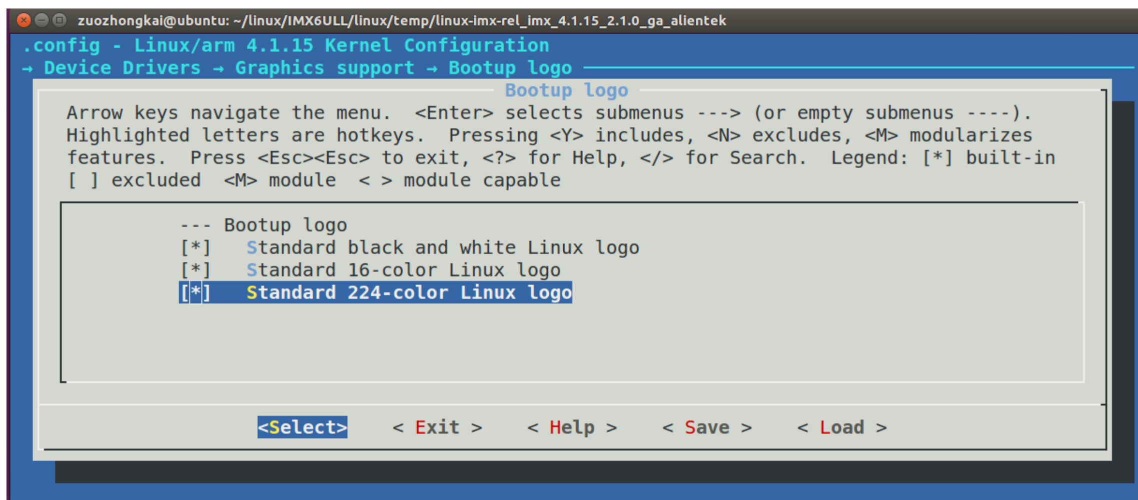


图 59.4.1.1 logo 配置项

图 59.4.1.1 中这三个选项分别对应黑白、16 位、24 位色彩格式的 logo，我们把这三个都选中，都编译进 Linux 内核里面。设置好以后保存退出，重新编译 Linux 内核，编译完成以后使用新编译出来的 imx6ull-alientek-emmc.dtb 和 zImage 镜像启动系统，如果 LCD 驱动工作正常的话就会在 LCD 屏幕左上角出现一个彩色的小企鹅 logo，屏幕背景色为黑色，如图 59.4.1.2 所示：

图 59.4.1.2 Linux 启动 logo 显示

59.4.2 设置 LCD 作为终端控制台

我们一直使用 SecureCRT 作为 Linux 开发板终端，开发板通过串口和 SecureCRT 进行通信。现在我们已经驱动起来 LCD 了，所以可以设置 LCD 作为终端，也就是开发板使用自己的显示设备作为自己的终端，然后接上键盘就可以直接在开发板上敲命令了，将 LCD 设置为终端控制台的方法如下：

1、设置 uboot 中的 bootargs

重启开发板，进入 Linux 命令行，重新设置 bootargs 参数的 console 内容，命令如下所示：

```
setenv bootargs 'console=tty1 console=ttymxc0,115200 root=/dev/nfs rw nfsroot=192.168.1.250:
/home/zuozhongkai/linux/nfs/rootfs ip=192.168.1.251:192.168.1.250:192.168.1.1:255.255.255.0::eth0:
off'
```

注意红色字体部分设置 console，这里我们设置了两遍 console，第一次设置 console=tty1，也就是设置 LCD 屏幕为控制台，第二遍又设置 console=ttymxc0,115200，也就是设置串口也作为控制台。相当于我们打开了两个 console，一个是 LCD，一个是串口，大家重启开发板就会发现 LCD 和串口都会显示 Linux 启动 log 信息。但是此时我们还不能使用 LCD 作为终端进行交互，因为我们的设置还未完成。

2、修改/etc/inittab 文件

打开开发板根文件系统中的/etc/inittab 文件，在里面加入下面这一行：

```
tty1::askfirst:-/bin/sh
```

添加完成以后的/etc/inittab 文件内容如图 59.4.2.1 所示：

```
#etc/inittab
::sysinit:/etc/init.d/rcS
console::askfirst:-/bin/sh
tty1::askfirst:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
::shutdown:/sbin/swapoff -a
```

打开tty1，也即是设置LCD作为终端

图 59.4.2.1 修改后的/etc/inittab 文件

修改完成以后保存/etc/inittab 并退出，然后重启开发板，重启以后开发板 LCD 屏幕最后一行会显示下面一行语句：

```
Please press Enter to activate this console.
```

上述提示语句说的是：按下回车键使能当前终端，我们在第五十八章已经将 I.MX6U-ALPHA 开发板上的 KEY 按键注册为了回车键，因此按下开发板上的 KEY 按键即可使能 LCD 这个终端。当然了，大家也可以接上一个 USB 键盘，Linux 内核默认已经使能了 USB 键盘驱动了，因此可以直接使用 USB 键盘。

至此，我们就拥有了两套终端，一个是基于串口的 SecureCRT，一个就是我们开发板的 LCD 屏幕，但是为了方便调试，我们以后还是以 SecureCRT 为主。我们可以通过下面这一行命令向 LCD 屏幕输出"hello linux!"

```
echo hello linux > /dev/tty1
```

59.4.3 LCD 背光调节

59.3 小节已经讲过了，背光设备树节点设置了 8 个等级的背光调节，可以设置为 0~7，我们可以通过设置背光等级来实现 LCD 背光亮度的调节，进入如下目录：

```
/sys/devices/platform/backlight/backlight/backlight
```

此目录下的文件如图 59.4.3.1 所示：

```
/sys/devices/platform/backlight/backlight/backlight # ls
actual_brightness   device             subsystem
bl_power            max_brightness     type
brightness          power              uevent
/sys/devices/platform/backlight/backlight/backlight #
```

图 59.4.3.1 目录下的文件和子目录

图 59.4.3.1 中的 brightness 表示当前亮度等级，max_bgigntness 表示最大亮度等级。当前这两个文件内容如图 59.4.3.2 所示：

```
/sys/devices/platform/backlight/backlight/backlight # cat max_brightness
7
/sys/devices/platform/backlight/backlight/backlight # cat brightness
6
/sys/devices/platform/backlight/backlight/backlight #
```

图 59.4.3.2 brightness 和 max_brightness 文件内容

从图 59.4.3.2 可以看出，当前屏幕亮度等级为 6，根据前面的分析可以，这个是 50%亮度。屏幕最大亮度等级为 7。如果我们要修改屏幕亮度，只需要向 brightness 写入需要设置的屏幕亮度等级即可。比如设置屏幕亮度等级为 7，那么可以使用如下命令：

```
echo 7 > brightness
```

输入上述命令以后就会发现屏幕亮度增大了，如果设置 brightness 为 0 的话就会关闭 LCD 背光，屏幕就会熄灭。

59.4.4 LCD 自动关闭解决方法

默认情况下 10 分钟以后 LCD 就会熄屏，这个并不是代码有问题，而是 Linux 内核设置的，就和我们用手机或者电脑一样，一段时间不操作的话屏幕就会熄灭，以节省电能。解决这个问题有多种方法，我们依次来看一下：

1、按键盘唤醒

最简单的就是按下回车键唤醒屏幕，我们在第 58 章将 I.MX6U-ALPHA 开发板上的 KEY 按键注册为了回车键，因此按下开发板上的 KEY 按键即可唤醒屏幕。如果开发板上没有按键的话可以外接 USB 键盘，然后按下 USB 键盘上的回车键唤醒屏幕。

2、关闭 10 分钟熄屏功能

在 Linux 源码中找到 drivers/tty/vt/vt.c 这个文件，在此文件中找到 blankinterval 变量，如下所示：

示例代码 59.4.4.1 blankinterval 变量

```
179 static int vesa_blank_mode;
180 static int vesa_off_interval;
181 static int blankinterval = 10*60;
```

blankinterval 变量控制着 LCD 关闭时间，默认是 10*60，也就是 10 分钟。将 blankinterval 的值改为 0 即可关闭 10 分钟熄屏的功能，修改完成以后需要重新编译 Linux 内核，得到新的 zImage，然后用新的 zImage 启动开发板。

3、编写一个 APP 来关闭熄屏功能

在 ubuntu 中新建一个名为 lcd_always_on.c 的文件，然后在里面输入如下所示内容：

示例代码 59.4.4.2 lcd_always_on.c 文件代码段

```
1  #include <fcntl.h>
2  #include <stdio.h>
3  #include <sys/ioctl.h>
4  
5  
6  int main(int argc, char *argv[])
7  {
8      int fd;
9      fd = open("/dev/tty1", O_RDWR);
10     write(fd, "\033[9;0]", 8);
11     close(fd);
12     return 0;
13 }
```

使用如下命令编译 lcd_always_on.c 这个文件：

```
arm-linux-gnueabihf-gcc lcd_always_on.c -o lcd_always_on
```

编译生成 lcd_always_on 以后将此可执行文件拷贝到开发板根文件系统的/usr/bin 目录中，然后给予可执行权限。设置 lcd_always_on 这个软件为开机自启动，打开/etc/init.d/rcS，在此文件最后面加入如下内容：

示例代码 59.4.4.3 lcd_always_on 自启动代码

```
1 cd /usr/bin
```

```
2 ./lcd_always_on
3 cd ..
```

修改完成以后保存/etc/init.d/rcS 文件，然后重启开发板即可。关于 Linux 下的 LCD 驱动我们就讲到这里。

第六十章 Linux RTC 驱动实验

RTC 也就是实时时钟，用于记录当前系统时间，对于 Linux 系统而言时间是非常重要的，就和我们使用 Windows 电脑或手机查看时间一样，我们在使用 Linux 设备的时候也需要查看时间。本章我们就来学习一下如何编写 Linux 下的 RTC 驱动程序。

60.1 Linux 内核 RTC 驱动简介

RTC 设备驱动是一个标准的字符设备驱动，应用程序通过 open、release、read、write 和 ioctl 等函数完成对 RTC 设备的操作，关于 RTC 硬件原理部分我们已经在裸机篇中的第二十五章进行了详细的讲解，这里就不再废话了。

Linux 内核将 RTC 设备抽象为 rtc_device 结构体，因此 RTC 设备驱动就是申请并初始化 rtc_device，最后将 rtc_device 注册到 Linux 内核里面，这样 Linux 内核就有一个 RTC 设备的。至于 RTC 设备的操作肯定是用一个操作集合(结构体)来表示的，我们先来看一下 rtc_device 结构体，此结构体定义在 include/linux/rtc.h 文件中，结构体内容如下(删除条件编译)：

示例代码 60.1.1 rtc_device 结构体

```
104 struct rtc_device
105 {
106     struct device dev;                  /* 设备              */
107     struct module *owner;
108 
109     int id;                             /* ID                */
110     char name[RTC_DEVICE_NAME_SIZE];     /* 名字              */
111 
112     const struct rtc_class_ops *ops;    /* RTC 设备底层操作函数 */
113     struct mutex ops_lock;
114 
115     struct cdev char_dev;               /* 字符设备          */
116     unsigned long flags;
117 
118     unsigned long irq_data;
119     spinlock_t irq_lock;
120     wait_queue_head_t irq_queue;
121     struct fasync_struct *async_queue;
122 
123     struct rtc_task *irq_task;
124     spinlock_t irq_task_lock;
125     int irq_freq;
126     int max_user_freq;
127 
128     struct timerqueue_head timerqueue;
129     struct rtc_timer aie_timer;
130     struct rtc_timer uie_rtctimer;
131     struct hrtimer pie_timer; /* sub second exp, so needs hrtimer */
132     int pie_enabled;
133     struct work_struct irqwork;
134     /* Some hardware can't support UIE mode */
135     int uie_unsupported;
......
```

```
147 };
```

我们需要重点关注的是 ops 成员变量，这是一个 rtc_class_ops 类型的指针变量，rtc_class_ops 为 RTC 设备的最底层操作函数集合，包括从 RTC 设备中读取时间、向 RTC 设备写入新的时间值等。因此，rtc_class_ops 是需要用户根据所使用的 RTC 设备编写的，此结构体定义在 include/linux/rtc.h 文件中，内容如下：

示例代码 60.1.2 rtc_class_ops 结构体

```
71 struct rtc_class_ops {
72  int (*open)(struct device *);
73  void (*release)(struct device *);
74  int (*ioctl)(struct device *, unsigned int, unsigned long);
75  int (*read_time)(struct device *, struct rtc_time *);
76  int (*set_time)(struct device *, struct rtc_time *);
77  int (*read_alarm)(struct device *, struct rtc_wkalrm *);
78  int (*set_alarm)(struct device *, struct rtc_wkalrm *);
79  int (*proc)(struct device *, struct seq_file *);
80  int (*set_mmss64)(struct device *, time64_t secs);
81  int (*set_mmss)(struct device *, unsigned long secs);
82  int (*read_callback)(struct device *, int data);
83  int (*alarm_irq_enable)(struct device *, unsigned int enabled);
84 };
```

看名字就知道 rtc_class_ops 操作集合中的这些函数是做什么的了，但是我们要注意，rtc_class_ops 中的这些函数只是最底层的 RTC 设备操作函数，并不是提供给应用层的 file_operations 函数操作集。RTC 是个字符设备，那么肯定有字符设备的 file_operations 函数操作集，Linux 内核提供了一个 RTC 通用字符设备驱动文件，文件名为 drivers/rtc/rtc-dev.c，rtc-dev.c 文件提供了所有 RTC 设备共用的 file_operations 函数操作集，如下所示：

示例代码 60.1.3 RTC 通用 file_operations 操作集

```
448 static const struct file_operations rtc_dev_fops = {
449     .owner      = THIS_MODULE,
450     .llseek     = no_llseek,
451     .read       = rtc_dev_read,
452     .poll       = rtc_dev_poll,
453     .unlocked_ioctl = rtc_dev_ioctl,
454     .open       = rtc_dev_open,
455     .release    = rtc_dev_release,
456     .fasync     = rtc_dev_fasync,
457 };
```

看到示例代码 60.1.3 是不是很熟悉了，标准的字符设备操作集。应用程序可以通过 ioctl 函数来设置/读取时间、设置/读取闹钟的操作，那么对应的 rtc_dev_ioctl 函数就会执行，rtc_dev_ioctl 最终会通过操作 rtc_class_ops 中的 read_time、set_time 等函数来对具体 RTC 设备的读写操作。我们简单来看一下 rtc_dev_ioctl 函数，函数内容如下(有省略)：

示例代码 60.1.4 rtc_dev_ioctl 函数代码段

```
218 static long rtc_dev_ioctl(struct file *file,
219         unsigned int cmd, unsigned long arg)
```

```
220 {
221     int err = 0;
222     struct rtc_device *rtc = file->private_data;
223     const struct rtc_class_ops *ops = rtc->ops;
224     struct rtc_time tm;
225     struct rtc_wkalrm alarm;
226     void __user *uarg = (void __user *) arg;
227 
228     err = mutex_lock_interruptible(&rtc->ops_lock);
229     if (err)
230         return err;
......
269     switch (cmd) {
......
333     case RTC_RD_TIME:              /* 读取时间 */
334         mutex_unlock(&rtc->ops_lock);
335 
336         err = rtc_read_time(rtc, &tm);
337         if (err < 0)
338             return err;
339 
340         if (copy_to_user(uarg, &tm, sizeof(tm)))
341             err = -EFAULT;
342         return err;
343 
344     case RTC_SET_TIME:          /* 设置时间 */
345         mutex_unlock(&rtc->ops_lock);
346 
347         if (copy_from_user(&tm, uarg, sizeof(tm)))
348             return -EFAULT;
349 
350         return rtc_set_time(rtc, &tm);
......
401     default:
402         /* Finally try the driver's ioctl interface */
403         if (ops->ioctl) {
404             err = ops->ioctl(rtc->dev.parent, cmd, arg);
405             if (err == -ENOIOCTLCMD)
406                 err = -ENOTTY;
407         } else
408             err = -ENOTTY;
409         break;
410     }
```

```
411
412 done:
413     mutex_unlock(&rtc->ops_lock);
414     return err;
415 }
```

第 333 行，RTC_RD_TIME 为时间读取命令。

第 336 行，如果是读取时间命令的话就调用 rtc_read_time 函数获取当前 RTC 时钟，rtc_read_time 函数，rtc_read_time 会调用__rtc_read_time 函数，__rtc_read_time 函数内容如下：

示例代码 60.1.5 __rtc_read_time 函数代码段

```
23 static int __rtc_read_time(struct rtc_device *rtc,
                               struct rtc_time *tm)
24 {
25     int err;
26     if (!rtc->ops)
27         err = -ENODEV;
28     else if (!rtc->ops->read_time)
29         err = -EINVAL;
30     else {
31         memset(tm, 0, sizeof(struct rtc_time));
32         err = rtc->ops->read_time(rtc->dev.parent, tm);
33         if (err < 0) {
34             dev_dbg(&rtc->dev, "read_time: fail to read: %d\n",
35                     err);
36             return err;
37         }
38
39         err = rtc_valid_tm(tm);
40         if (err < 0)
41             dev_dbg(&rtc->dev, "read_time: rtc_time isn't valid\n");
42     }
43     return err;
44 }
```

从示例代码 60.1.5 中的 32 行可以看出，__rtc_read_time 函数会通过调用 rtc_class_ops 中的 read_time 来从 RTC 设备中获取当前时间。rtc_dev_ioctl 函数对其他的命令处理都是类似的，比如 RTC_ALM_READ 命令会通过 rtc_read_alarm 函数获取到闹钟值，而 rtc_read_alarm 函数经过层层调用，最终会调用 rtc_class_ops 中的 read_alarm 函数来获取闹钟值。

至此，Linux 内核中 RTC 驱动调用流程就很清晰了，如图 60.1.1 所示：

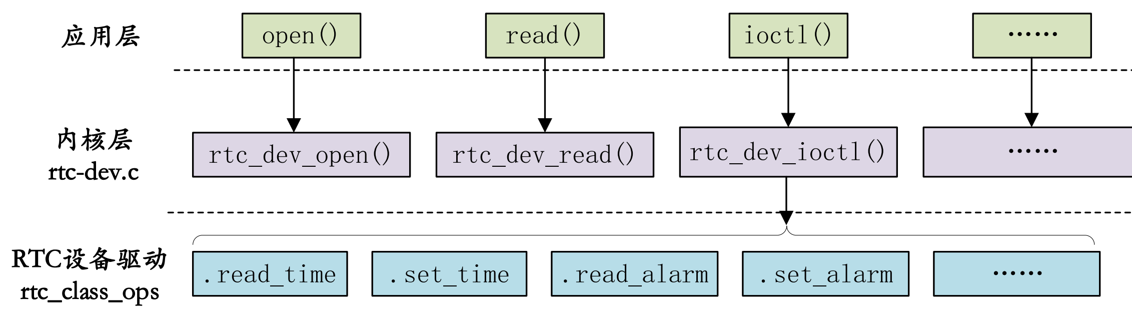


图 60.1.1 Linux RTC 驱动调用流程

当 rtc_class_ops 准备好以后需要将其注册到 Linux 内核中，这里我们可以使用 rtc_device_register 函数完成注册工作。此函数会申请一个 rtc_device 并且初始化这个 rtc_device，最后向调用者返回这个 rtc_device，此函数原型如下：

```
struct rtc_device *rtc_device_register(const char              *name,
                                       struct device           *dev,
                                       const struct rtc_class_ops *ops,
                                       struct module           *owner)
```

函数参数和返回值含义如下：

name：设备名字。

dev：设备。

ops：RTC 底层驱动函数集。

owner：驱动模块拥有者。

返回值：注册成功的话就返回 rtc_device，错误的话会返回一个负值。

当卸载 RTC 驱动的时候需要调用 rtc_device_unregister 函数来注销注册的 rtc_device，函数原型如下：

```
void rtc_device_unregister(struct rtc_device    *rtc)
```

函数参数和返回值含义如下：

rtc：要删除的 rtc_device。

返回值：无。

还有另外一对 rtc_device 注册函数 devm_rtc_device_register 和 devm_rtc_device_unregister，分别为注册和注销 rtc_device。

60.2 I.MX6U 内部 RTC 驱动分析

先直接告诉大家，I.MX6U 的 RTC 驱动我们不用自己编写，因为 NXP 已经写好了。其实对于大多数的 SOC 来讲，内部 RTC 驱动都不需要我们去编写，半导体厂商会编写好。但是这不代表我们就偷懒了，虽然不用编写 RTC 驱动，但是我们得看一下这些原厂是怎么编写 RTC 驱动的。

分析驱动，先从设备树入手，打开 imx6ull.dtsi，在里面找到如下 snvs_rtc 设备节点，节点内容如下所示：

示例代码 60.2.1 imx6ull.dtsi 文件 rtc 设备节点

```
1 snvs_rtc: snvs-rtc-lp {
2     compatible = "fsl,sec-v4.0-mon-rtc-lp";
3     regmap = <&snvs>;
4     offset = <0x34>;
```

```
5     interrupts = <GIC_SPI 19 IRQ_TYPE_LEVEL_HIGH>, <GIC_SPI 20
                    IRQ_TYPE_LEVEL_HIGH>;
6 };
```

第 2 行设置兼容属性 compatible 的值为“fsl,sec-v4.0-mon-rtc-lp”，因此在 Linux 内核源码中搜索此字符串即可找到对应的驱动文件，此文件为 drivers/rtc/rtc-snvs.c，在 rtc-snvs.c 文件中找到如下所示内容：

示例代码 60.2.2 rtc 设备 platform 驱动框架

```
380 static const struct of_device_id snvs_dt_ids[] = {
381     { .compatible = "fsl,sec-v4.0-mon-rtc-lp", },
382     { /* sentinel */ }
383 };
384 MODULE_DEVICE_TABLE(of, snvs_dt_ids);
385 
386 static struct platform_driver snvs_rtc_driver = {
387     .driver = {
388         .name   = "snvs_rtc",
389         .pm = SNVS_RTC_PM_OPS,
390         .of_match_table = snvs_dt_ids,
391     },
392     .probe      = snvs_rtc_probe,
393 };
394 module_platform_driver(snvs_rtc_driver);
```

第 380~383 行，设备树 ID 表，有一条 compatible 属性，值为“fsl,sec-v4.0-mon-rtc-lp”，因此 imx6ull.dtsi 中的 snvs_rtc 设备节点会和此驱动匹配。

第 386~393 行，标准的 platform 驱动框架，当设备和驱动匹配成功以后 snvs_rtc_probe 函数就会执行。我们来看一下 snvs_rtc_probe 函数，函数内容如下(有省略)：

示例代码 60.2.3 snvs_rtc_probe 函数代码段

```
238 static int snvs_rtc_probe(struct platform_device *pdev)
239 {
240     struct snvs_rtc_data *data;
241     struct resource *res;
242     int ret;
243     void __iomem *mmio;
244 
245     data = devm_kzalloc(&pdev->dev, sizeof(*data), GFP_KERNEL);
246     if (!data)
247         return -ENOMEM;
248 
249     data->regmap = 
           syscon_regmap_lookup_by_phandle(pdev->dev.of_node, "regmap");
250 
251     if (IS_ERR(data->regmap)) {
252         dev_warn(&pdev->dev, "snvs rtc: you use old dts file,
```

```
                  please update it\n");
253         res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
254 
255         mmio = devm_ioremap_resource(&pdev->dev, res);
256         if (IS_ERR(mmio))
257             return PTR_ERR(mmio);
258 
259         data->regmap = devm_regmap_init_mmio(&pdev->dev, mmio,
                                                &snvs_rtc_config);
260     } else {
261         data->offset = SNVS_LPREGISTER_OFFSET;
262         of_property_read_u32(pdev->dev.of_node, "offset",
                                      &data->offset);
263     }
264 
265     if (!data->regmap) {
266         dev_err(&pdev->dev, "Can't find snvs syscon\n");
267         return -ENODEV;
268     }
269 
270     data->irq = platform_get_irq(pdev, 0);
271     if (data->irq < 0)
272         return data->irq;
......
285 
286     platform_set_drvdata(pdev, data);
287 
288     /* Initialize glitch detect */
289     regmap_write(data->regmap, data->offset + SNVS_LPPGDR,
                  SNVS_LPPGDR_INIT);
290 
291     /* Clear interrupt status */
292     regmap_write(data->regmap, data->offset + SNVS_LPSR,
                     0xffffffff);
293 
294     /* Enable RTC */
295     snvs_rtc_enable(data, true);
296 
297     device_init_wakeup(&pdev->dev, true);
298 
299     ret = devm_request_irq(&pdev->dev, data->irq,
                              snvs_rtc_irq_handler,
300                           IRQF_SHARED, "rtc alarm", &pdev->dev);
```

```
301     if (ret) {
302         dev_err(&pdev->dev, "failed to request irq %d: %d\n",
303             data->irq, ret);
304         goto error_rtc_device_register;
305     }
306 
307     data->rtc = devm_rtc_device_register(&pdev->dev, pdev->name,
308                 &snvs_rtc_ops, THIS_MODULE);
309     if (IS_ERR(data->rtc)) {
310         ret = PTR_ERR(data->rtc);
311         dev_err(&pdev->dev, "failed to register rtc: %d\n", ret);
312         goto error_rtc_device_register;
313     }
314 
315     return 0;
316 
317 error_rtc_device_register:
318     if (data->clk)
319         clk_disable_unprepare(data->clk);
320 
321     return ret;
322 }
```

第 253 行，调用 platform_get_resource 函数从设备树中获取到 RTC 外设寄存器基地址。

第 255 行，调用函数 devm_ioremap_resource 完成内存映射，得到 RTC 外设寄存器物理基地址对应的虚拟地址。

第 259 行，Linux3.1 引入了一个全新的 regmap 机制，regmap 用于提供一套方便的 API 函数去操作底层硬件寄存器，以提高代码的可重用性。snvs-rtc.c 文件会采用 regmap 机制来读写 RTC 底层硬件寄存器。这里使用 devm_regmap_init_mmio 函数将 RTC 的硬件寄存器转化为 regmap 形式，这样 regmap 机制的 regmap_write、regmap_read 等 API 函数才能操作寄存器。

第 270 行，从设备树中获取 RTC 的中断号。

第 289 行，设置 RTC_LPPGDR 寄存器值为 SNVS_LPPGDR_INIT= 0x41736166，这里就是用的 regmap 机制的 regmap_write 函数完成对寄存器进行写操作。

第 292 行，设置 RTC_LPSR 寄存器，写入 0xffffffff，LPSR 是 RTC 状态寄存器，写 1 清零，因此这一步就是清除 LPSR 寄存器。

第 295 行，调用 snvs_rtc_enable 函数使能 RTC，此函数会设置 RTC_LPCR 寄存器。

第 299 行，调用 devm_request_irq 函数请求 RTC 中断，中断服务函数为 snvs_rtc_irq_handler，用于 RTC 闹钟中断。

第 307 行，调用 devm_rtc_device_register 函数向系统注册 rtc_devcie，RTC 底层驱动集为 snvs_rtc_ops。snvs_rtc_ops 操作集包含了读取/设置 RTC 时间，读取/设置闹钟等函数。snvs_rtc_ops 内容如下：

示例代码 60.2.4 snvs_rtc_ops 操作集

```
200 static const struct rtc_class_ops snvs_rtc_ops = {
201     .read_time = snvs_rtc_read_time,
```

```
202     .set_time = snvs_rtc_set_time,
203     .read_alarm = snvs_rtc_read_alarm,
204     .set_alarm = snvs_rtc_set_alarm,
205     .alarm_irq_enable = snvs_rtc_alarm_irq_enable,
206 };
```

我们就以第 201 行的 snvs_rtc_read_time 函数为例讲解一下 rtc_class_ops 的各个 RTC 底层操作函数该如何去编写。snvs_rtc_read_time 函数用于读取 RTC 时间值，此函数内容如下所示：

示例代码 60.2.5 snvs_rtc_read_time 函数代码段

```
126 static int snvs_rtc_read_time(struct device *dev,
                                 struct rtc_time *tm)
127 {
128     struct snvs_rtc_data *data = dev_get_drvdata(dev);
129     unsigned long time = rtc_read_lp_counter(data);
130 
131     rtc_time_to_tm(time, tm);
132 
133     return 0;
134 }
```

第 129 行，调用 rtc_read_lp_counter 获取 RTC 计数值，这个时间值是秒数。

第 131 行，调用 rtc_time_to_tm 函数将获取到的秒数转换为时间值，也就是 rtc_time 结构体类型，rtc_time 结构体定义如下：

示例代码 60.2.6 rtc_time 结构体类型

```
20 struct rtc_time {
21     int tm_sec;
22     int tm_min;
23     int tm_hour;
24     int tm_mday;
25     int tm_mon;
26     int tm_year;
27     int tm_wday;
28     int tm_yday;
29     int tm_isdst;
30 };
```

最后我们来看一下 rtc_read_lp_counter 函数，此函数用于读取 RTC 计数值，函数内容如下(有省略)：

示例代码 60.2.7 rtc_read_lp_counter 函数代码段

```
50 static u32 rtc_read_lp_counter(struct snvs_rtc_data *data)
51 {
52     u64 read1, read2;
53     u32 val;
54 
55     do {
56         regmap_read(data->regmap, data->offset + SNVS_LPSRTCMR,
```

```
                    &val);
57          read1 = val;
58          read1 <<= 32;
59          regmap_read(data->regmap, data->offset + SNVS_LPSRTCLR,
                    &val);
60          read1 |= val;
61
62          regmap_read(data->regmap, data->offset + SNVS_LPSRTCMR,
                  &val);
63          read2 = val;
64          read2 <<= 32;
65          regmap_read(data->regmap, data->offset + SNVS_LPSRTCLR,
                    &val);
66          read2 |= val;
67      /*
68      * when CPU/BUS are running at low speed, there is chance that
69      * we never get same value during two consecutive read, so here
70      * we only compare the second value.
71      */
72      } while ((read1 >> CNTR_TO_SECS_SH) != (read2 >>
             CNTR_TO_SECS_SH));
73
74      /* Convert 47-bit counter to 32-bit raw second count */
75      return (u32) (read1 >> CNTR_TO_SECS_SH);
76 }
```

第 56~72 行，读取 RTC_LPSRTCMR 和 RTC_LPSRTCLR 这两个寄存器，得到 RTC 的计数值，单位为秒，这个秒数就是当前时间。这里读取了两次 RTC 计数值，因为要读取两个寄存器，因此可能存在读取第二个寄存器的时候时间数据更新了，导致时间不匹配，因此这里连续读两次，如果两次的时间值相等那么就表示时间数据有效。

第 75 行，返回时间值，注意这里将前面读取到的 RTC 计数值右移了 15 位。

这个就是 snvs_rtc_read_time 函数读取 RTC 时间值的过程，至于其他的底层操作函数大家自行分析即可，都是大同小异的，这里就不再分析了。关于 I.MX6U 内部 RTC 驱动源码就讲解到这里。

60.3 RTC 时间查看与设置

1、时间 RTC 查看

RTC 是用来计时的，因此最基本的就是查看时间，Linux 内核启动的时候可以看到系统时钟设置信息，如图 60.3.1 所示：

```
snvs_rtc 20cc000.snvs:snvs-rtc-lp: rtc core: registered 20cc000.snvs:snvs-r as rtc0
i2c /dev entries driver
IR NEC protocol handler initialized
IR RC5(x/sz) protocol handler initialized
```
将snvs-rtc-lp设置为rtc0

图 60.3.1 Linux 启动 log 信息

从图 60.3.1 中可以看出，Linux 内核在启动的时候将 snvs_rtc 设置为 rtc0，大家的启动信息可能会和图 60.3.1 中的不同，但是内容基本上都是一样的。

如果要查看时间的话输入“date”命令即可，结果如图 60.3.2 所示：

```
/ # date
Thu Jan  1 00:06:11 UTC 1970
/ #
```

图 60.3.2 当前时间值

从图 60.3.2 可以看出，当前时间为 1970 年 1 月 1 日 00:06:11，很明显是时间不对，我们需要重新设置 RTC 时间。

2、设置 RTC 时间

RTC 时间设置也是使用的 date 命令，输入“date --help”命令即可查看 date 命令如何设置系统时间，结果如图 60.3.3 所示：

```
/ # date --help
BusyBox v1.29.0 (2019-06-13 11:08:23 CST) multi-call binary.

Usage: date [OPTIONS] [+FMT] [TIME]

Display time (using +FMT), or set time

        [-s,--set] TIME Set time to TIME
        -u,--utc        Work in UTC (don't convert to local time)
        -R,--rfc-2822   Output RFC-2822 compliant date string
        -I[SPEC]        Output ISO-8601 compliant date string
                        SPEC='date' (default) for date only,
                        'hours', 'minutes', or 'seconds' for date and
                        time to the indicated precision
        -r,--reference FILE     Display last modification time of FILE
        -d,--date TIME  Display TIME, not 'now'
        -D FMT          Use FMT for -d TIME conversion

Recognized TIME formats:
        hh:mm[:ss]
        [YYYY.]MM.DD-hh:mm[:ss]
        YYYY-MM-DD hh:mm[:ss]
        [[[[[YY]YY]MM]DD]hh]mm[.ss]
        'date TIME' form accepts MMDDhhmm[[YY]YY][.ss] instead
```

图 60.3.3 date 命令帮助信息

现在我要设置当前时间为 2019 年 8 月 31 日 18:13:00，因此输入如下命令：

```
date -s "2019-08-31 18:13:00"
```

设置完成以后再次使用 date 命令查看一下当前时间就会发现时间改过来了，如图 60.3.4 所示：

```
/ # date
Sat Aug 31 18:13:01 UTC 2019
/ #
```

图 60.3.4 当前时间

大家注意我们使用“date -s”命令仅仅是将当前系统时间设置了，此时间还没有写入到 I.MX6U 内部 RTC 里面或其他的 RTC 芯片里面，因此系统重启以后时间又会丢失。我们需要将当前的时间写入到 RTC 里面，这里要用到 hwclock 命令，输入如下命令将系统时间写入到 RTC 里面：

```
hwclock -w          //将当前系统时间写入到 RTC 里面
```

时间写入到 RTC 里面以后就不怕系统重启以后时间丢失了，如果 I.MX6U-ALPHA 开发板底板接了纽扣电池，那么开发板即使断电了时间也不会丢失。大家可以尝试一下不断电重启和

断电重启这两种情况下开发板时间会不会丢失。

第六十一章 Linux I2C 驱动实验

I2C 是很常用的一个串行通信接口，用于连接各种外设、传感器等器件，在裸机篇已经对 I.MX6U 的 I2C 接口做了详细的讲解。本章我们来学习一下如何在 Linux 下开发 I2C 接口器件驱动，重点是学习 Linux 下的 I2C 驱动框架，按照指定的框架去编写 I2C 设备驱动。本章同样以 I.MX6U-ALPHA 开发板上的 AP3216C 这个三合一环境光传感器为例，通过 AP3216C 讲解一下如何编写 Linux 下的 I2C 设备驱动程序。

61.1 Linux I2C 驱动框架简介

回想一下我们在裸机篇中是怎么编写 AP3216C 驱动的，我们编写了四个文件：bsp_i2c.c、bsp_i2c.h、bsp_ap3216c.c 和 bsp_ap3216c.h。其中前两个是 I.MX6U 的 IIC 接口驱动，后两个文件是 AP3216C 这个 I2C 设备驱动文件。相当于有两部分驱动：

①、I2C 主机驱动。

②、I2C 设备驱动。

对于 I2C 主机驱动，一旦编写完成就不需要再做修改，其他的 I2C 设备直接调用主机驱动提供的 API 函数完成读写操作即可。这个正好符合 Linux 的驱动分离与分层的思想，因此 Linux 内核也将 I2C 驱动分为两部分：

①、I2C 总线驱动，I2C 总线驱动就是 SOC 的 I2C 控制器驱动，也叫做 I2C 适配器驱动。

②、I2C 设备驱动，I2C 设备驱动就是针对具体的 I2C 设备而编写的驱动。

61.1.1 I2C 总线驱动

首先来看一下 I2C 总线，在讲 platform 的时候就说过，platform 是虚拟出来的一条总线，目的是为了实现总线、设备、驱动框架。对于 I2C 而言，不需要虚拟出一条总线，直接使用 I2C 总线即可。I2C 总线驱动重点是 I2C 适配器(也就是 SOC 的 I2C 接口控制器)驱动，这里要用到两个重要的数据结构：i2c_adapter 和 i2c_algorithm，Linux 内核将 SOC 的 I2C 适配器(控制器)抽象成 i2c_adapter，i2c_adapter 结构体定义在 include/linux/i2c.h 文件中，结构体内容如下：

示例代码 61.1.1.1 i2c_adapter 结构体

```
498 struct i2c_adapter {
499     struct module *owner;
500     unsigned int class;       /* classes to allow probing for */
501     const struct i2c_algorithm *algo; /* 总线访问算法 */
502     void *algo_data;
503 
504     /* data fields that are valid for all devices   */
505     struct rt_mutex bus_lock;
506 
507     int timeout;           /* in jiffies */
508     int retries;
509     struct device dev;      /* the adapter device */
510 
511     int nr;
512     char name[48];
513     struct completion dev_released;
514 
515     struct mutex userspace_clients_lock;
516     struct list_head userspace_clients;
517 
518     struct i2c_bus_recovery_info *bus_recovery_info;
519     const struct i2c_adapter_quirks *quirks;
520 };
```

第 501 行，i2c_algorithm 类型的指针变量 algo，对于一个 I2C 适配器，肯定要对外提供读写 API 函数，设备驱动程序可以使用这些 API 函数来完成读写操作。i2c_algorithm 就是 I2C 适配器与 IIC 设备进行通信的方法。

i2c_algorithm 结构体定义在 include/linux/i2c.h 文件中，内容如下(删除条件编译)：

示例代码 61.1.1.2 i2c_algorithm 结构体

```
391 struct i2c_algorithm {
......
398     int (*master_xfer)(struct i2c_adapter *adap,
                           struct i2c_msg *msgs,
399                 int num);
400     int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
401                 unsigned short flags, char read_write,
402                 u8 command, int size, union i2c_smbus_data *data);
403
404     /* To determine what the adapter supports */
405     u32 (*functionality) (struct i2c_adapter *);
......
411 };
```

第 398 行，master_xfer 就是 I2C 适配器的传输函数，可以通过此函数来完成与 IIC 设备之间的通信。

第 400 行，smbus_xfer 就是 SMBUS 总线的传输函数。

综上所述，I2C 总线驱动，或者说 I2C 适配器驱动的主要工作就是初始化 i2c_adapter 结构体变量，然后设置 i2c_algorithm 中的 master_xfer 函数。完成以后通过 i2c_add_numbered_adapter 或 i2c_add_adapter 这两个函数向系统注册设置好的 i2c_adapter，这两个函数的原型如下：

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
```

这两个函数的区别在于 i2c_add_adapter 使用动态的总线号，而 i2c_add_numbered_adapter 使用静态总线号。函数参数和返回值含义如下：

adapter 或 adap：要添加到 Linux 内核中的 i2c_adapter，也就是 I2C 适配器。

返回值：0，成功；负值，失败。

如果要删除 I2C 适配器的话使用 i2c_del_adapter 函数即可，函数原型如下：

```
void i2c_del_adapter(struct i2c_adapter * adap)
```

函数参数和返回值含义如下：

adap：要删除的 I2C 适配器。

返回值：无。

关于 I2C 的总线(控制器或适配器)驱动就讲解到这里，一般 SOC 的 I2C 总线驱动都是由半导体厂商编写的，比如 I.MX6U 的 I2C 适配器驱动 NXP 已经编写好了，这个不需要用户去编写。因此 I2C 总线驱动其实跟我们这些 SOC 使用者来说是被屏蔽掉的，我们只要专注于 I2C 设备驱动即可。除非你是在半导体公司上班，工作内容就是写 I2C 适配器驱动。

61.1.2 I2C 设备驱动

I2C 设备驱动重点关注两个数据结构：i2c_client 和 i2c_driver，根据总线、设备和驱动模型，I2C 总线上一小节已经讲了。还剩下设备和驱动，i2c_client 就是描述设备信息的，i2c_driver 描

述驱动内容，类似于 platform_driver。

1、i2c_client 结构体

i2c_client 结构体定义在 include/linux/i2c.h 文件中，内容如下：

示例代码 61.1.2.1 i2c_client 结构体

```
217 struct i2c_client {
218     unsigned short flags;           /* 标志                    */
219     unsigned short addr;            /* 芯片地址，7 位，存在低 7 位*/
......
222     char name[I2C_NAME_SIZE];       /* 名字                    */
223     struct i2c_adapter *adapter;    /* 对应的 I2C 适配器         */
224     struct device dev;              /* 设备结构体               */
225     int irq;                        /* 中断                    */
226     struct list_head detected;
......
230 };
```

一个设备对应一个 i2c_client，每检测到一个 I2C 设备就会给这个 I2C 设备分配一个 i2c_client。

2、i2c_driver 结构体

i2c_driver 类似 platform_driver，是我们编写 I2C 设备驱动重点要处理的内容，i2c_driver 结构体定义在 include/linux/i2c.h 文件中，内容如下：

示例代码 61.1.2.2 i2c_driver 结构体

```
161 struct i2c_driver {
162     unsigned int class;
163 
164     /* Notifies the driver that a new bus has appeared. You should
165      * avoid  using this, it will be removed in a near future.
166      */
167     int (*attach_adapter)(struct i2c_adapter *) __deprecated;
168 
169     /* Standard driver model interfaces */
170     int (*probe)(struct i2c_client *, const struct i2c_device_id *);
171     int (*remove)(struct i2c_client *);
172 
173     /* driver model interfaces that don't relate to enumeration  */
174     void (*shutdown)(struct i2c_client *);
175 
176     /* Alert callback, for example for the SMBus alert protocol.
177      * The format and meaning of the data value depends on the
178      * protocol.For the SMBus alert protocol, there is a single bit
179      * of data passed  as the alert response's low bit ("event
180      flag"). */
181     void (*alert)(struct i2c_client *, unsigned int data);
```

```
182
183     /* a ioctl like command that can be used to perform specific
184      * functions with the device.
185      */
186     int (*command)(struct i2c_client *client, unsigned int cmd,
                              void *arg);
187
188     struct device_driver driver;
189     const struct i2c_device_id *id_table;
190
191     /* Device detection callback for automatic device creation */
192     int (*detect)(struct i2c_client *, struct i2c_board_info *);
193     const unsigned short *address_list;
194     struct list_head clients;
195 };
```

第 170 行，当 I2C 设备和驱动匹配成功以后 probe 函数就会执行，和 platform 驱动一样。

第 188 行，device_driver 驱动结构体，如果使用设备树的话，需要设置 device_driver 的 of_match_table 成员变量，也就是驱动的兼容(compatible)属性。

第 189 行，id_table 是传统的、未使用设备树的设备匹配 ID 表。

对于我们 I2C 设备驱动编写人来说，重点工作就是构建 i2c_driver，构建完成以后需要向 Linux 内核注册这个 i2c_driver。i2c_driver 注册函数为 int i2c_register_driver，此函数原型如下：

```
int i2c_register_driver(struct module      *owner,
                        struct i2c_driver  *driver)
```

函数参数和返回值含义如下：

owner：一般为 THIS_MODULE。

driver：要注册的 i2c_driver。

返回值：0，成功；负值，失败。

另外 i2c_add_driver 也常常用于注册 i2c_driver，i2c_add_driver 是一个宏，定义如下：

示例代码 61.1.2.3 i2c_add_driver 宏

```
587 #define i2c_add_driver(driver) \
588     i2c_register_driver(THIS_MODULE, driver)
```

i2c_add_driver 就是对 i2c_register_driver 做了一个简单的封装，只有一个参数，就是要注册的 i2c_driver。

注销 I2C 设备驱动的时候需要将前面注册的 i2c_driver 从 Linux 内核中注销掉，需要用到 i2c_del_driver 函数，此函数原型如下：

```
void i2c_del_driver(struct i2c_driver    *driver)
```

函数参数和返回值含义如下：

driver：要注销的 i2c_driver。

返回值：无。

i2c_driver 的注册示例代码如下：

示例代码 61.1.2.4 i2c_driver 注册流程

```
1  /* i2c 驱动的 probe 函数 */
2  static int xxx_probe(struct i2c_client *client,
```

```
                    const struct i2c_device_id *id)
3  {
4      /* 函数具体程序 */
5      return 0;
6  }
7
8  /* i2c 驱动的 remove 函数 */
9  static int ap3216c_remove(struct i2c_client *client)
10 {
11     /* 函数具体程序 */
12     return 0;
13 }
14
15 /* 传统匹配方式 ID 列表 */
16 static const struct i2c_device_id xxx_id[] = {
17     {"xxx", 0},
18     {}
19 };
20
21 /* 设备树匹配列表 */
22 static const struct of_device_id xxx_of_match[] = {
23     { .compatible = "xxx" },
24     { /* Sentinel */ }
25 };
26
27 /* i2c 驱动结构体 */
28 static struct i2c_driver xxx_driver = {
29     .probe = xxx_probe,
30     .remove = xxx_remove,
31     .driver = {
32         .owner = THIS_MODULE,
33         .name = "xxx",
34         .of_match_table = xxx_of_match,
35         },
36         .id_table = xxx_id,
37     };
38
39 /* 驱动入口函数 */
40 static int __init xxx_init(void)
41 {
42     int ret = 0;
43
44     ret = i2c_add_driver(&xxx_driver);
```

```
45     return ret;
46 }
47
48 /* 驱动出口函数 */
49 static void __exit xxx_exit(void)
50 {
51     i2c_del_driver(&xxx_driver);
52 }
53
54 module_init(xxx_init);
55 module_exit(xxx_exit);
```

第 16~19 行，i2c_device_id，无设备树的时候匹配 ID 表。

第 22~25 行，of_device_id，设备树所使用的匹配表。

第 28~37 行，i2c_driver，当 I2C 设备和 I2C 驱动匹配成功以后 probe 函数就会执行，这些和 platform 驱动一样，probe 函数里面基本就是标准的字符设备驱动那一套了。

61.1.3 I2C 设备和驱动匹配过程

I2C 设备和驱动的匹配过程是由 I2C 核心来完成的，drivers/i2c/i2c-core.c 就是 I2C 的核心部分，I2C 核心提供了一些与具体硬件无关的 API 函数，比如前面讲过的：

1、i2c_adapter 注册/注销函数

int i2c_add_adapter(struct i2c_adapter *adapter)

int i2c_add_numbered_adapter(struct i2c_adapter *adap)

void i2c_del_adapter(struct i2c_adapter * adap)

2、i2c_driver 注册/注销函数

int i2c_register_driver(struct module *owner, struct i2c_driver *driver)

int i2c_add_driver (struct i2c_driver *driver)

void i2c_del_driver(struct i2c_driver *driver)

设备和驱动的匹配过程也是由 I2C 总线完成的，I2C 总线的数据结构为 i2c_bus_type，定义在 drivers/i2c/i2c-core.c 文件，i2c_bus_type 内容如下：

示例代码 61.1.2.5 i2c_bus_type 总线

```
736 struct bus_type i2c_bus_type = {
737     .name       = "i2c",
738     .match      = i2c_device_match,
739     .probe      = i2c_device_probe,
740     .remove     = i2c_device_remove,
741     .shutdown   = i2c_device_shutdown,
742 };
```

.match 就是 I2C 总线的设备和驱动匹配函数，在这里就是 i2c_device_match 这个函数，此函数内容如下：

示例代码 61.1.2.6 i2c_device_match 函数

```
457 static int i2c_device_match(struct device *dev, struct 
device_driver *drv)
```

```
458 {
459     struct i2c_client   *client = i2c_verify_client(dev);
460     struct i2c_driver   *driver;
461
462     if (!client)
463         return 0;
464
465     /* Attempt an OF style match */
466     if (of_driver_match_device(dev, drv))
467         return 1;
468
469     /* Then ACPI style match */
470     if (acpi_driver_match_device(dev, drv))
471         return 1;
472
473     driver = to_i2c_driver(drv);
474     /* match on an id table if there is one */
475     if (driver->id_table)
476         return i2c_match_id(driver->id_table, client) != NULL;
477
478     return 0;
479 }
```

第 466 行，of_driver_match_device 函数用于完成设备树设备和驱动匹配。比较 I2C 设备节点的 compatible 属性和 of_device_id 中的 compatible 属性是否相等，如果相当的话就表示 I2C 设备和驱动匹配。

第 470 行，acpi_driver_match_device 函数用于 ACPI 形式的匹配。

第 476 行，i2c_match_id 函数用于传统的、无设备树的 I2C 设备和驱动匹配过程。比较 I2C 设备名字和 i2c_device_id 的 name 字段是否相等，相等的话就说明 I2C 设备和驱动匹配。

61.2 I.MX6U 的 I2C 适配器驱动分析

上一小节我们讲解了 Linux 下的 I2C 驱动框架，重点分为 I2C 适配器驱动和 I2C 设备驱动，其中 I2C 适配器驱动就是 SOC 的 I2C 控制器驱动。I2C 设备驱动是需要用户根据不同的 I2C 设备去编写，而 I2C 适配器驱动一般都是 SOC 厂商去编写的，比如 NXP 就编写好了 I.MX6U 的 I2C 适配器驱动。在 imx6ull.dtsi 文件中找到 I.MX6U 的 I2C1 控制器节点，节点内容如下所示：

示例代码 61.2.1 I2C1 控制器节点

```
1 i2c1: i2c@021a0000 {
2   #address-cells = <1>;
3   #size-cells = <0>;
4   compatible = "fsl,imx6ul-i2c", "fsl,imx21-i2c";
5   reg = <0x021a0000 0x4000>;
6   interrupts = <GIC_SPI 36 IRQ_TYPE_LEVEL_HIGH>;
7   clocks = <&clks IMX6UL_CLK_I2C1>;
8   status = "disabled";
```

```
9 };
```

重点关注 i2c1 节点的 compatible 属性值，因为通过 compatible 属性值可以在 Linux 源码里面找到对应的驱动文件。这里 i2c1 节点的 compatible 属性值有两个："fsl,imx6ul-i2c"和"fsl,imx21-i2c"，在 Linux 源码中搜索这两个字符串即可找到对应的驱动文件。I.MX6U 的 I2C 适配器驱动驱动文件为 drivers/i2c/busses/i2c-imx.c，在此文件中有如下内容：

示例代码 61.2.2 i2c-imx.c 文件代码段

```
244 static struct platform_device_id imx_i2c_devtype[] = {
245     {
246         .name = "imx1-i2c",
247         .driver_data = (kernel_ulong_t)&imx1_i2c_hwdata,
248     }, {
249         .name = "imx21-i2c",
250         .driver_data = (kernel_ulong_t)&imx21_i2c_hwdata,
251     }, {
252         /* sentinel */
253     }
254 };
255 MODULE_DEVICE_TABLE(platform, imx_i2c_devtype);
256 
257 static const struct of_device_id i2c_imx_dt_ids[] = {
258     { .compatible = "fsl,imx1-i2c", .data = &imx1_i2c_hwdata, },
259     { .compatible = "fsl,imx21-i2c", .data = &imx21_i2c_hwdata, },
260     { .compatible = "fsl,vf610-i2c", .data = &vf610_i2c_hwdata, },
261     { /* sentinel */ }
262 };
263 MODULE_DEVICE_TABLE(of, i2c_imx_dt_ids);
......
1119 static struct platform_driver i2c_imx_driver = {
1120    .probe = i2c_imx_probe,
1121    .remove = i2c_imx_remove,
1122    .driver = {
1123        .name = DRIVER_NAME,
1124        .owner = THIS_MODULE,
1125        .of_match_table = i2c_imx_dt_ids,
1126        .pm = IMX_I2C_PM,
1127    },
1128    .id_table   = imx_i2c_devtype,
1129 };
1130 
1131 static int __init i2c_adap_imx_init(void)
1132 {
1133    return platform_driver_register(&i2c_imx_driver);
1134 }
```

```
1135 subsys_initcall(i2c_adap_imx_init);
1136
1137 static void __exit i2c_adap_imx_exit(void)
1138 {
1139     platform_driver_unregister(&i2c_imx_driver);
1140 }
1141 module_exit(i2c_adap_imx_exit);
```

从示例代码 61.2.2 可以看出，I.MX6U 的 I2C 适配器驱动是个标准的 platform 驱动，由此可以看出，虽然 I2C 总线为别的设备提供了一种总线驱动框架，但是 I2C 适配器却是 platform 驱动。就像你的部门老大是你的领导，你是他的下属，但是放到整个公司，你的部门老大却也是老板的下属。

第 259 行，“fsl,imx21-i2c”属性值，设备树中 i2c1 节点的 compatible 属性值就是与此匹配上的。因此 i2c-imx.c 文件就是 I.MX6U 的 I2C 适配器驱动文件。

第 1120 行，当设备和驱动匹配成功以后 i2c_imx_probe 函数就会执行，i2c_imx_probe 函数就会完成 I2C 适配器初始化工作。

i2c_imx_probe 函数内容如下所示(有省略)：

示例代码 61.2.3 i2c_imx_probe 函数代码段

```
971  static int i2c_imx_probe(struct platform_device *pdev)
972  {
973      const struct of_device_id *of_id =
974                  of_match_device(i2c_imx_dt_ids, &pdev->dev);
975      struct imx_i2c_struct *i2c_imx;
976      struct resource *res;
977      struct imxi2c_platform_data *pdata =
                     dev_get_platdata(&pdev->dev);
978      void __iomem *base;
979      int irq, ret;
980      dma_addr_t phy_addr;
981
982      dev_dbg(&pdev->dev, "<%s>\n", __func__);
983
984      irq = platform_get_irq(pdev, 0);
......
990      res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
991      base = devm_ioremap_resource(&pdev->dev, res);
992      if (IS_ERR(base))
993          return PTR_ERR(base);
994
995      phy_addr = (dma_addr_t)res->start;
996      i2c_imx = devm_kzalloc(&pdev->dev, sizeof(*i2c_imx),
                     GFP_KERNEL);
997      if (!i2c_imx)
998          return -ENOMEM;
```

```
999
1000    if (of_id)
1001        i2c_imx->hwdata = of_id->data;
1002    else
1003        i2c_imx->hwdata = (struct imx_i2c_hwdata *)
1004                platform_get_device_id(pdev)->driver_data;
1005
1006    /* Setup i2c_imx driver structure */
1007    strlcpy(i2c_imx->adapter.name, pdev->name,
                 sizeof(i2c_imx->adapter.name));
1008    i2c_imx->adapter.owner     = THIS_MODULE;
1009    i2c_imx->adapter.algo      = &i2c_imx_algo;
1010    i2c_imx->adapter.dev.parent = &pdev->dev;
1011    i2c_imx->adapter.nr     = pdev->id;
1012    i2c_imx->adapter.dev.of_node    = pdev->dev.of_node;
1013    i2c_imx->base          = base;
1014
1015    /* Get I2C clock */
1016    i2c_imx->clk = devm_clk_get(&pdev->dev, NULL);
......
1022    ret = clk_prepare_enable(i2c_imx->clk);
......
1027    /* Request IRQ */
1028    ret = devm_request_irq(&pdev->dev, irq, i2c_imx_isr,
1029                 IRQF_NO_SUSPEND, pdev->name, i2c_imx);
......
1035    /* Init queue */
1036    init_waitqueue_head(&i2c_imx->queue);
1037
1038    /* Set up adapter data */
1039    i2c_set_adapdata(&i2c_imx->adapter, i2c_imx);
1040
1041    /* Set up clock divider */
1042    i2c_imx->bitrate = IMX_I2C_BIT_RATE;
1043    ret = of_property_read_u32(pdev->dev.of_node,
1044                 "clock-frequency", &i2c_imx->bitrate);
1045    if (ret < 0 && pdata && pdata->bitrate)
1046        i2c_imx->bitrate = pdata->bitrate;
1047
1048    /* Set up chip registers to defaults */
1049    imx_i2c_write_reg(i2c_imx->hwdata->i2cr_ien_opcode ^ I2CR_IEN,
1050            i2c_imx, IMX_I2C_I2CR);
1051    imx_i2c_write_reg(i2c_imx->hwdata->i2sr_clr_opcode, i2c_imx,
```

```
                              IMX_I2C_I2SR);
1052
1053    /* Add I2C adapter */
1054    ret = i2c_add_numbered_adapter(&i2c_imx->adapter);
1055    if (ret < 0) {
1056        dev_err(&pdev->dev, "registration failed\n");
1057        goto clk_disable;
1058    }
1059
1060    /* Set up platform driver data */
1061    platform_set_drvdata(pdev, i2c_imx);
1062    clk_disable_unprepare(i2c_imx->clk);
......
1070    /* Init DMA config if supported */
1071    i2c_imx_dma_request(i2c_imx, phy_addr);
1072
1073    return 0;   /* Return OK */
1074
1075 clk_disable:
1076    clk_disable_unprepare(i2c_imx->clk);
1077    return ret;
1078 }
```

第 984 行，调用 platform_get_irq 函数获取中断号。

第 990~991 行，调用 platform_get_resource 函数从设备树中获取 I2C1 控制器寄存器物理基地址，也就是 0X021A0000。获取到寄存器基地址以后使用 devm_ioremap_resource 函数对其进行内存映射，得到可以在 Linux 内核中使用的虚拟地址。

第 996 行，NXP 使用 imx_i2c_struct 结构体来表示 I.MX 系列 SOC 的 I2C 控制器，这里使用 devm_kzalloc 函数来申请内存。

第 1008~1013 行，imx_i2c_struct 结构体要有个叫做 adapter 的成员变量，adapter 就是 i2c_adapter，这里初始化 i2c_adapter。第 1009 行设置 i2c_adapter 的 algo 成员变量为 i2c_imx_algo，也就是设置 i2c_algorithm。

第 1028~1029 行，注册 I2C 控制器中断，中断服务函数为 i2c_imx_isr。

第 1042~1044 行，设置 I2C 频率默认为 IMX_I2C_BIT_RATE=100KHz，如果设备树节点设置了“clock-frequency”属性的话 I2C 频率就使用 clock-frequency 属性值。

第 1049~1051 行，设置 I2C1 控制的 I2CR 和 I2SR 寄存器。

第 1054 行，调用 i2c_add_numbered_adapter 函数向 Linux 内核注册 i2c_adapter。

第 1071 行，申请 DMA，看来 I.MX 的 I2C 适配器驱动采用了 DMA 方式。

i2c_imx_probe 函数主要的工作就是一下两点：

①、初始化 i2c_adapter，设置 i2c_algorithm 为 i2c_imx_algo，最后向 Linux 内核注册 i2c_adapter。

②、初始化 I2C1 控制器的相关寄存器。

i2c_imx_algo 包含 I2C1 适配器与 I2C 设备的通信函数 master_xfer，i2c_imx_algo 结构体定义如下：

示例代码 61.2.4 i2c_imx_algo 结构体

```
966 static struct i2c_algorithm i2c_imx_algo = {
967     .master_xfer    = i2c_imx_xfer,
968     .functionality  = i2c_imx_func,
969 };
```

我们先来看一下. functionality，functionality 用于返回此 I2C 适配器支持什么样的通信协议，在这里 functionality 就是 i2c_imx_func 函数，i2c_imx_func 函数内容如下：

示例代码 61.2.5 i2c_imx_func 函数

```
static u32 i2c_imx_func(struct i2c_adapter *adapter)
{
    return I2C_FUNC_I2C | I2C_FUNC_SMBUS_EMUL
        | I2C_FUNC_SMBUS_READ_BLOCK_DATA;
}
```

重点来看一下 i2c_imx_xfer 函数，因为最终就是通过此函数来完成与 I2C 设备通信的，此函数内容如下(有省略)：

示例代码 61.2.6 i2c_imx_xfer 函数

```
888 static int i2c_imx_xfer(struct i2c_adapter *adapter,
889                         struct i2c_msg *msgs, int num)
890 {
891     unsigned int i, temp;
892     int result;
893     bool is_lastmsg = false;
894     struct imx_i2c_struct *i2c_imx = i2c_get_adapdata(adapter);
895 
896     dev_dbg(&i2c_imx->adapter.dev, "<%s>\n", __func__);
897 
898     /* Start I2C transfer */
899     result = i2c_imx_start(i2c_imx);
900     if (result)
901         goto fail0;
902 
903     /* read/write data */
904     for (i = 0; i < num; i++) {
905         if (i == num - 1)
906             is_lastmsg = true;
907 
908         if (i) {
909             dev_dbg(&i2c_imx->adapter.dev,
910                 "<%s> repeated start\n", __func__);
911             temp = imx_i2c_read_reg(i2c_imx, IMX_I2C_I2CR);
912             temp |= I2CR_RSTA;
913             imx_i2c_write_reg(temp, i2c_imx, IMX_I2C_I2CR);
914             result =  i2c_imx_bus_busy(i2c_imx, 1);
```

```
915             if (result)
916                 goto fail0;
917         }
918         dev_dbg(&i2c_imx->adapter.dev,
919             "<%s> transfer message: %d\n", __func__, i);
920         /* write/read data */
......
938         if (msgs[i].flags & I2C_M_RD)
939             result = i2c_imx_read(i2c_imx, &msgs[i], is_lastmsg);
940         else {
941             if (i2c_imx->dma && msgs[i].len >= DMA_THRESHOLD)
942                 result = i2c_imx_dma_write(i2c_imx, &msgs[i]);
943             else
944                 result = i2c_imx_write(i2c_imx, &msgs[i]);
945         }
946         if (result)
947             goto fail0;
948     }
949
950 fail0:
951     /* Stop I2C transfer */
952     i2c_imx_stop(i2c_imx);
953
954     dev_dbg(&i2c_imx->adapter.dev, "<%s> exit with: %s: %d\n",
__func__,
955         (result < 0) ? "error" : "success msg",
956             (result < 0) ? result : num);
957     return (result < 0) ? result : num;
958 }
```

第 899 行，调用 i2c_imx_start 函数开启 I2C 通信。

第 939 行，如果是从 I2C 设备读数据的话就调用 i2c_imx_read 函数。

第 941~945 行，向 I2C 设备写数据，如果要用 DMA 的话就使用 i2c_imx_dma_write 函数来完成写数据。如果不使用 DMA 的话就使用 i2c_imx_write 函数完成写数据。

第 952 行，I2C 通信完成以后调用 i2c_imx_stop 函数停止 I2C 通信。

i2c_imx_start、i2c_imx_read、i2c_imx_write 和 i2c_imx_stop 这些函数就是 I2C 寄存器的具体操作函数，函数内容基本和我们裸机篇中讲的 I2C 驱动一样，这里我们就不详细的分析了，大家可以对照着第二十六章实验自行分析。

61.3 I2C 设备驱动编写流程

I2C 适配器驱动 SOC 厂商已经替我们编写好了，我们需要做的就是编写具体的设备驱动，本小节我们就来学习一下 I2C 设备驱动的详细编写流程。

61.3.1 I2C 设备信息描述

1、未使用设备树的时候

首先肯定要描述 I2C 设备节点信息，先来看一下没有使用设备树的时候是如何在 BSP 里面描述 I2C 设备信息的，在未使用设备树的时候需要在 BSP 里面使用 i2c_board_info 结构体来描述一个具体的 I2C 设备。i2c_board_info 结构体如下：

示例代码 61.3.1.1 i2c_board_info 结构体

```
295 struct i2c_board_info {
296     char        type[I2C_NAME_SIZE];     /* I2C 设备名字 */
297     unsigned short  flags;               /* 标志         */
298     unsigned short  addr;                /* I2C 器件地址 */
299     void        *platform_data;
300     struct dev_archdata *archdata;
301     struct device_node *of_node;
302     struct fwnode_handle *fwnode;
303     int     irq;
304 };
```

type 和 addr 这两个成员变量是必须要设置的，一个是 I2C 设备的名字，一个是 I2C 设备的器件地址。打开 arch/arm/mach-imx/mach-mx27_3ds.c 文件，此文件中关于 OV2640 的 I2C 设备信息描述如下：

示例代码 61.3.1.2 OV2640 的 I2C 设备信息

```
392 static struct i2c_board_info mx27_3ds_i2c_camera = {
393     I2C_BOARD_INFO("ov2640", 0x30),
394 };
```

示例代码 61.3.1.2 中使用 I2C_BOARD_INFO 来完成 mx27_3ds_i2c_camera 的初始化工作，I2C_BOARD_INFO 是一个宏，定义如下：

示例代码 61.3.1.3 I2C_BOARD_INFO 宏

```
316 #define I2C_BOARD_INFO(dev_type, dev_addr) \
317     .type = dev_type, .addr = (dev_addr)
```

可以看出，I2C_BOARD_INFO 宏其实就是设置 i2c_board_info 的 type 和 addr 这两个成员变量，因此示例代码 61.3.1.2 的主要工作就是设置 I2C 设备名字为 ov2640，ov2640 的器件地址为 0X30。

大家可以在 Linux 源码里面全局搜索 i2c_board_info，会找到大量以 i2c_board_info 定义的 I2C 设备信息，这些就是未使用设备树的时候 I2C 设备的描述方式，当采用了设备树以后就不会再使用 i2c_board_info 来描述 I2C 设备了。

2、使用设备树的时候

使用设备树的时候 I2C 设备信息通过创建相应的节点就行了，比如 NXP 官方的 EVK 开发板在 I2C1 上接了 mag3110 这个磁力计芯片，因此必须在 i2c1 节点下创建 mag3110 子节点，然后在这个子节点内描述 mag3110 这个芯片的相关信息。打开 imx6ull-14x14-evk.dts 这个设备树文件，然后找到如下内容：

示例代码 61.3.1.4 mag3110 子节点

```
1  &i2c1 {
```

```
2       clock-frequency = <100000>;
3       pinctrl-names = "default";
4       pinctrl-0 = <&pinctrl_i2c1>;
5       status = "okay";
6
7       mag3110@0e {
8           compatible = "fsl,mag3110";
9           reg = <0x0e>;
10          position = <2>;
11      };
......
20 };
```

第 7~11 行，向 i2c1 添加 mag3110 子节点，第 7 行“mag3110@0e”是子节点名字，“@”后面的“0e”就是 mag3110 的 I2C 器件地址。第 8 行设置 compatible 属性值为“fsl,mag3110”。第 9 行的 reg 属性也是设置 mag3110 的器件地址的，因此值为 0x0e。I2C 设备节点的创建重点是 compatible 属性和 reg 属性的设置，一个用于匹配驱动，一个用于设置器件地址。

61.3.2 I2C 设备数据收发处理流程

在 61.1.2 小节已经说过了，I2C 设备驱动首先要做的就是初始化 i2c_driver 并向 Linux 内核注册。当设备和驱动匹配以后 i2c_driver 里面的 probe 函数就会执行，probe 函数里面所做的就是字符设备驱动那一套了。一般需要在 probe 函数里面初始化 I2C 设备，要初始化 I2C 设备就必须能够对 I2C 设备寄存器进行读写操作，这里就要用到 i2c_transfer 函数了。i2c_transfer 函数最终会调用 I2C 适配器中 i2c_algorithm 里面的 master_xfer 函数，对于 I.MX6U 而言就是 i2c_imx_xfer 这个函数。i2c_transfer 函数原型如下：

```
int i2c_transfer(struct i2c_adapter     *adap,
                 struct i2c_msg         *msgs,
                 int                    num)
```

函数参数和返回值含义如下：

adap： 所使用的 I2C 适配器，i2c_client 会保存其对应的 i2c_adapter。

msgs： I2C 要发送的一个或多个消息。

num： 消息数量，也就是 msgs 的数量。

返回值： 负值，失败，其他非负值，发送的 msgs 数量。

我们重点来看一下 msgs 这个参数，这是一个 i2c_msg 类型的指针参数，I2C 进行数据收发说白了就是消息的传递，Linux 内核使用 i2c_msg 结构体来描述一个消息。i2c_msg 结构体定义在 include/uapi/linux/i2c.h 文件中，结构体内容如下：

示例代码 61.3.2.1 i2c_msg 结构体

```
68 struct i2c_msg {
69      __u16 addr;                 /*  从机地址        */
70      __u16 flags;                /*  标志            */
71      #define I2C_M_TEN           0x0010
72      #define I2C_M_RD            0x0001
73      #define I2C_M_STOP          0x8000
74      #define I2C_M_NOSTART       0x4000
```

```
75     #define I2C_M_REV_DIR_ADDR  0x2000
76     #define I2C_M_IGNORE_NAK    0x1000
77     #define I2C_M_NO_RD_ACK     0x0800
78     #define I2C_M_RECV_LEN      0x0400
79     __u16 len;                  /* 消息(本 msg)长度 */
80     __u8 *buf;                  /* 消息数据        */
81 };
```

使用 i2c_transfer 函数发送数据之前要先构建好 i2c_msg，使用 i2c_transfer 进行 I2C 数据收发的示例代码如下：

示例代码 61.3.2.2 I2C 设备多寄存器数据读写

```
1  /* 设备结构体 */
2  struct xxx_dev {
3      ......
4      void *private_data; /* 私有数据，一般会设置为 i2c_client */
5  };
6
7  /*
8   * @description    : 读取 I2C 设备多个寄存器数据
9   * @param - dev    : I2C 设备
10  * @param - reg    : 要读取的寄存器首地址
11  * @param - val    : 读取到的数据
12  * @param - len    : 要读取的数据长度
13  * @return         : 操作结果
14  */
15 static int xxx_read_regs(struct xxx_dev *dev, u8 reg, void *val,
                         int len)
16 {
17     int ret;
18     struct i2c_msg msg[2];
19     struct i2c_client *client = (struct i2c_client *)
                                   dev->private_data;
20
21     /* msg[0]，第一条写消息，发送要读取的寄存器首地址 */
22     msg[0].addr = client->addr;         /* I2C 器件地址     */
23     msg[0].flags = 0;                   /* 标记为发送数据   */
24     msg[0].buf = &reg;                  /* 读取的首地址     */
25     msg[0].len = 1;                     /* reg 长度         */
26
27     /* msg[1]，第二条读消息，读取寄存器数据 */
28     msg[1].addr = client->addr;         /* I2C 器件地址     */
29     msg[1].flags = I2C_M_RD;            /* 标记为读取数据   */
30     msg[1].buf = val;                   /* 读取数据缓冲区   */
31     msg[1].len = len;                   /* 要读取的数据长度 */
```

```
32 
33     ret = i2c_transfer(client->adapter, msg, 2);
34     if(ret == 2) {
35         ret = 0;
36     } else {
37         ret = -EREMOTEIO;
38     }
39     return ret;
40 }
41 
42 /*
43  * @description    : 向 I2C 设备多个寄存器写入数据
44  * @param - dev    : 要写入的设备结构体
45  * @param - reg    : 要写入的寄存器首地址
46  * @param - val    : 要写入的数据缓冲区
47  * @param - len    : 要写入的数据长度
48  * @return         : 操作结果
49  */
50 static s32 xxx_write_regs(struct xxx_dev *dev, u8 reg, u8 *buf,
                           u8 len)
51 {
52     u8 b[256];
53     struct i2c_msg msg;
54     struct i2c_client *client = (struct i2c_client *)
                                 dev->private_data;
55 
56     b[0] = reg;                 /* 寄存器首地址                    */
57     memcpy(&b[1],buf,len);      /* 将要发送的数据拷贝到数组 b 里面 */
58 
59     msg.addr = client->addr;    /* I2C 器件地址                    */
60     msg.flags = 0;              /* 标记为写数据                    */
61 
62     msg.buf = b;                /* 要发送的数据缓冲区              */
63     msg.len = len + 1;          /* 要发送的数据长度                */
64 
65     return i2c_transfer(client->adapter, &msg, 1);
66 }
```

第 2~5 行，设备结构体，在设备结构体里面添加一个执行 void 的指针成员变量 private_data，此成员变量用于保存设备的私有数据。在 I2C 设备驱动中我们一般将其指向 I2C 设备对应的 i2c_client。

第 15~40 行，xxx_read_regs 函数用于读取 I2C 设备多个寄存器数据。第 18 行定义了一个 i2c_msg 数组，2 个数组元素，因为 I2C 读取数据的时候要先发送要读取的寄存器地址，然后再读取数据，所以需要准备两个 i2c_msg。一个用于发送寄存器地址，一个用于读取寄存器值。对

于 msg[0]，将 flags 设置为 0，表示写数据。msg[0]的 addr 是 I2C 设备的器件地址，msg[0]的 buf 成员变量就是要读取的寄存器地址。对于 msg[1]，将 flags 设置为 I2C_M_RD，表示读取数据。msg[1]的 buf 成员变量用于保存读取到的数据，len 成员变量就是要读取的数据长度。调用 i2c_transfer 函数完成 I2C 数据读操作。

第 50~66 行，xxx_write_regs 函数用于向 I2C 设备多个寄存器写数据，I2C 写操作要比读操作简单一点，因此一个 i2c_msg 即可。数组 b 用于存放寄存器首地址和要发送的数据，第 59 行设置 msg 的 addr 为 I2C 器件地址。第 60 行设置 msg 的 flags 为 0，也就是写数据。第 62 行设置要发送的数据，也就是数组 b。第 63 行设置 msg 的 len 为 len+1，因为要加上一个字节的寄存器地址。最后通过 i2c_transfer 函数完成向 I2C 设备的写操作。

另外还有两个 API 函数分别用于 I2C 数据的收发操作，这两个函数最终都会调用 i2c_transfer。首先来看一下 I2C 数据发送函数 i2c_master_send，函数原型如下：

```
int i2c_master_send(const struct i2c_client   *client,
                    const char                *buf,
                    int                       count)
```

函数参数和返回值含义如下：

client：I2C 设备对应的 i2c_client。

buf：要发送的数据。

count：要发送的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值：负值，失败，其他非负值，发送的字节数。

I2C 数据接收函数为 i2c_master_recv，函数原型如下：

```
int i2c_master_recv(const struct i2c_client   *client,
                    char                      *buf,
                    int                       count)
```

函数参数和返回值含义如下：

client：I2C 设备对应的 i2c_client。

buf：要接收的数据。

count：要接收的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值：负值，失败，其他非负值，发送的字节数。

关于 Linux 下 I2C 设备驱动的编写流程就讲解到这里，重点就是 i2c_msg 的构建和 i2c_transfer 函数的调用，接下来我们就编写 AP3216C 这个 I2C 设备的 Linux 驱动。

61.4 硬件原理图分析

本章实验硬件原理图参考 26.2 小节即可。

61.5 实验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 21_iic。

61.5.1 修改设备树

1、IO 修改或添加

首先肯定是要修改 IO，AP3216C 用到了 I2C1 接口，I.MX6U-ALPHA 开发板上的 I2C1 接口使用到了 UART4_TXD 和 UART4_RXD，因此肯定要在设备树里面设置这两个 IO。如果要用到 AP3216C 的中断功能的话还需要初始化 AP_INT 对应的 GIO1_IO01 这个 IO，本章实验我们不使用中断功能。因此只需要设置 UART4_TXD 和 UART4_RXD 这两个 IO，NXP 其实已经将他这两个 IO 设置好了，打开 imx6ull-alientek-emmc.dts，然后找到如下内容：

示例代码 61.5.1.1 pinctrl_i2c1 子节点

```
1 pinctrl_i2c1: i2c1grp {
2     fsl,pins = <
3         MX6UL_PAD_UART4_TX_DATA__I2C1_SCL 0x4001b8b0
4         MX6UL_PAD_UART4_RX_DATA__I2C1_SDA 0x4001b8b0
5     >;
6 };
```

pinctrl_i2c1 就是 I2C1 的 IO 节点，这里将 UART4_TXD 和 UART4_RXD 这两个 IO 分别复用为 I2C1_SCL 和 I2C1_SDA，电气属性都设置为 0x4001b8b0。

2、在 i2c1 节点追加 ap3216c 子节点

AP3216C 是连接到 I2C1 上的，因此需要在 i2c1 节点下添加 ap3216c 的设备子节点，在 imx6ull-alientek-emmc.dts 文件中找到 i2c1 节点，此节点默认内容如下：

示例代码 61.5.1.2 i2c1 子节点默认内容

```
1  &i2c1 {
2      clock-frequency = <100000>;
3      pinctrl-names = "default";
4      pinctrl-0 = <&pinctrl_i2c1>;
5      status = "okay";
6  
7      mag3110@0e {
8          compatible = "fsl,mag3110";
9          reg = <0x0e>;
10         position = <2>;
11     };
12 
13     fxls8471@1e {
14         compatible = "fsl,fxls8471";
15         reg = <0x1e>;
16         position = <0>;
17         interrupt-parent = <&gpio5>;
18         interrupts = <0 8>;
19     };
20 };
```

第 2 行，clock-frequency 属性为 I2C 频率，这里设置为 100KHz。

第 4 行，pinctrl-0 属性指定 I2C 所使用的 IO 为示例代码 61.5.1.1 中的 pinctrl_i2c1 子节点。

第 7~11 行，mag3110 是个磁力计，NXP 官方的 EVK 开发板上接了 mag3110，因此 NXP 在 i2c1 节点下添加了 mag3110 这个子节点。正点原子的 I.MX6U-ALPHA 开发板上没有用到 mag3110，因此需要将此节点删除掉。

第 13~19 行，NXP 官方 EVK 开发板也接了一个 fxls8471，正点原子的 I.MX6U-ALPHA 开发板同样没有此器件，所以也要将其删除掉。

将 i2c1 节点里面原有的 mag3110 和 fxls8471 这两个 I2C 子节点删除，然后添加 ap3216c 子节点信息，完成以后的 i2c1 节点内容如下所示：

示例代码 61.5.1.3 添加 ap3216c 子节点以后的 i2c1 节点

```
1  &i2c1 {
2      clock-frequency = <100000>;
3      pinctrl-names = "default";
4      pinctrl-0 = <&pinctrl_i2c1>;
5      status = "okay";
6  
7      ap3216c@1e {
8          compatible = "alientek,ap3216c";
9          reg = <0x1e>;
10     };
11 };
```

第 7 行，ap3216c 子节点，@后面的“1e”是 ap3216c 的器件地址。

第 8 行，设置 compatible 值为“alientek,ap3216c”。

第 9 行，reg 属性也是设置 ap3216c 器件地址的，因此 reg 设置为 0x1e。

设备树修改完成以后使用“make dtbs”重新编译一下，然后使用新的设备树启动 Linux 内核。/sys/bus/i2c/devices 目录下存放着所有 I2C 设备，如果设备树修改正确的话，会在/sys/bus/i2c/devices 目录下看到一个名为“0-001e”的子目录，如图 61.5.1.1 所示：

```
/sys/bus/i2c/devices # ls
0-001e   1-001a   1-003c   i2c-0    i2c-1
```

图 61.5.1.1 当前系统 I2C 设备

图 61.5.1.1 中的“0-001e”就是 ap3216c 的设备目录，“1e”就是 ap3216c 器件地址。进入 0-001e 目录，可以看到“name”文件，name 问价就保存着此设备名字，在这里就是“ap3216c”，如图 61.5.1.2 所示：

```
/sys/bus/i2c/devices # cat 0-001e/name
ap3216c
/sys/bus/i2c/devices #
```

图 61.5.1.2 ap3216c 器件名字

61.5.2 AP3216C 驱动编写

新建名为“21_iic”的文件夹，然后在 21_iic 文件夹里面创建 vscode 工程，工作区命名为“iic”。工程创建好以后新建 ap3216c.c 和 ap3216creg.h 这两个文件，ap3216c.c 为 AP3216C 的驱动代码，ap3216creg.h 是 AP3216C 寄存器头文件。先在 ap3216creg.h 中定义好 AP3216C 的寄存器，输入如下内容，

示例代码 61.5.2.1 ap3216creg.h 文件代码段

```
1  #ifndef AP3216C_H
2  #define AP3216C_H
3  /***************************************************************
4  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
5  文件名        : ap3216creg.h
6  作者          : 左忠凯
7  版本          : V1.0
8  描述          : AP3216C 寄存器地址描述头文件
9  其他          : 无
10 论坛          : www.openedv.com
11 日志          : 初版 V1.0 2019/9/2 左忠凯创建
12 ***************************************************************/
13 /* AP3316C 寄存器 */
14 #define AP3216C_SYSTEMCONG   0x00   /* 配置寄存器       */
15 #define AP3216C_INTSTATUS    0X01   /* 中断状态寄存器   */
16 #define AP3216C_INTCLEAR     0X02   /* 中断清除寄存器   */
17 #define AP3216C_IRDATALOW    0x0A   /* IR 数据低字节    */
18 #define AP3216C_IRDATAHIGH   0x0B   /* IR 数据高字节    */
19 #define AP3216C_ALSDATALOW   0x0C   /* ALS 数据低字节   */
20 #define AP3216C_ALSDATAHIGH  0X0D   /* ALS 数据高字节   */
21 #define AP3216C_PSDATALOW    0X0E   /* PS 数据低字节    */
22 #define AP3216C_PSDATAHIGH   0X0F   /* PS 数据高字节    */
23
24 #endif
```

ap3216creg.h 没什么好讲的，就是一些寄存器宏定义。然后在 ap3216c.c 输入如下内容：

示例代码 61.5.2.2 ap3216c.c 文件代码段

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of_gpio.h>
12 #include <linux/semaphore.h>
13 #include <linux/timer.h>
14 #include <linux/i2c.h>
15 #include <asm/mach/map.h>
16 #include <asm/uaccess.h>
17 #include <asm/io.h>
```

```
18 #include "ap3216creg.h"
19 /***************************************************************
20 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
21 文件名      : ap3216c.c
22 作者        : 左忠凯
23 版本        : V1.0
24 描述        : AP3216C 驱动程序
25 其他        : 无
26 论坛        : www.openedv.com
27 日志        : 初版 V1.0 2019/9/2 左忠凯创建
28 ***************************************************************/
29 #define AP3216C_CNT      1
30 #define AP3216C_NAME     "ap3216c"
31 
32 struct ap3216c_dev {
33     dev_t devid;                /* 设备号         */
34     struct cdev cdev;           /* cdev           */
35     struct class *class;        /* 类             */
36     struct device *device;      /* 设备           */
37     struct device_node  *nd;    /* 设备节点       */
38     int major;                  /* 主设备号       */
39     void *private_data;         /* 私有数据       */
40     unsigned short ir, als, ps; /* 三个光传感器数据 */
41 };
42 
43 static struct ap3216c_dev ap3216cdev;
44 
45 /*
46  * @description   : 从 ap3216c 读取多个寄存器数据
47  * @param - dev   :  ap3216c 设备
48  * @param - reg   :  要读取的寄存器首地址
49  * @param - val   :  读取到的数据
50  * @param - len   :  要读取的数据长度
51  * @return        : 操作结果
52  */
53 static int ap3216c_read_regs(struct ap3216c_dev *dev, u8 reg,
                              void *val, int len)
54 {
55     int ret;
56     struct i2c_msg msg[2];
57     struct i2c_client *client = (struct i2c_client *)
                                   dev->private_data;
58 
```

```
59      /* msg[0]为发送要读取的首地址 */
60      msg[0].addr = client->addr;      /* ap3216c 地址    */
61      msg[0].flags = 0;                /* 标记为发送数据   */
62      msg[0].buf = &reg;               /* 读取的首地址     */
63      msg[0].len = 1;                  /* reg 长度        */
64
65      /* msg[1]读取数据 */
66      msg[1].addr = client->addr;      /* ap3216c 地址    */
67      msg[1].flags = I2C_M_RD;         /* 标记为读取数据   */
68      msg[1].buf = val;                /* 读取数据缓冲区   */
69      msg[1].len = len;                /* 要读取的数据长度 */
70
71      ret = i2c_transfer(client->adapter, msg, 2);
72      if(ret == 2) {
73          ret = 0;
74      } else {
75          printk("i2c rd failed=%d reg=%06x len=%d\n",ret, reg, len);
76          ret = -EREMOTEIO;
77      }
78      return ret;
79  }
80
81  /*
82   * @description   : 向 ap3216c 多个寄存器写入数据
83   * @param - dev   :  ap3216c 设备
84   * @param - reg   :  要写入的寄存器首地址
85   * @param - val   :  要写入的数据缓冲区
86   * @param - len   :  要写入的数据长度
87   * @return        :   操作结果
88   */
89  static s32 ap3216c_write_regs(struct ap3216c_dev *dev, u8 reg,
                                  u8 *buf, u8 len)
90  {
91      u8 b[256];
92      struct i2c_msg msg;
93      struct i2c_client *client = (struct i2c_client *)
                                        dev->private_data;
94
95      b[0] = reg;                  /* 寄存器首地址                  */
96      memcpy(&b[1],buf,len);       /* 将要写入的数据拷贝到数组 b 里面 */
97
98      msg.addr = client->addr;     /* ap3216c 地址                 */
99      msg.flags = 0;               /* 标记为写数据                   */
```

```
100
101     msg.buf = b;                /* 要写入的数据缓冲区          */
102     msg.len = len + 1;          /* 要写入的数据长度            */
103
104     return i2c_transfer(client->adapter, &msg, 1);
105 }
106
107 /*
108  * @description  : 读取 ap3216c 指定寄存器值，读取一个寄存器
109  * @param - dev  :  ap3216c 设备
110  * @param - reg  :  要读取的寄存器
111  * @return       :   读取到的寄存器值
112  */
113 static unsigned char ap3216c_read_reg(struct ap3216c_dev *dev,
                                              u8 reg)
114 {
115     u8 data = 0;
116
117     ap3216c_read_regs(dev, reg, &data, 1);
118     return data;
119
120 #if 0
121     struct i2c_client *client = (struct i2c_client *)
                                   dev->private_data;
122     return i2c_smbus_read_byte_data(client, reg);
123 #endif
124 }
125
126 /*
127  * @description  : 向 ap3216c 指定寄存器写入指定的值，写一个寄存器
128  * @param - dev  :  ap3216c 设备
129  * @param - reg  :  要写的寄存器
130  * @param - data : 要写入的值
131  * @return       :    无
132  */
133 static void ap3216c_write_reg(struct ap3216c_dev *dev, u8 reg,
                                 u8 data)
134 {
135     u8 buf = 0;
136     buf = data;
137     ap3216c_write_regs(dev, reg, &buf, 1);
138 }
139
```

```
140 /*
141  * @description   : 读取 AP3216C 的数据，读取原始数据，包括 ALS,PS 和 IR,
142  *                 :同时打开 ALS,IR+PS 的话两次数据读取的间隔要大于 112.5ms
143  * @param - ir     : ir 数据
144  * @param - ps     : ps 数据
145  * @param - ps     : als 数据
146  * @return         : 无。
147  */
148 void ap3216c_readdata(struct ap3216c_dev *dev)
149 {
150     unsigned char i =0;
151     unsigned char buf[6];
152 
153     /* 循环读取所有传感器数据 */
154     for(i = 0; i < 6; i++)
155     {
156         buf[i] = ap3216c_read_reg(dev, AP3216C_IRDATALOW + i);
157     }
158 
159     if(buf[0] & 0X80)     /* IR_OF 位为 1,则数据无效      */
160         dev->ir = 0;
161     else                  /* 读取 IR 传感器的数据          */
162         dev->ir = ((unsigned short)buf[1] << 2) | (buf[0] & 0X03);
163 
164     dev->als = ((unsigned short)buf[3] << 8) | buf[2];/* ALS 数据 */
165 
166     if(buf[4] & 0x40)     /* IR_OF 位为 1,则数据无效      */
167         dev->ps = 0;
168     else                  /* 读取 PS 传感器的数据          */
169         dev->ps = ((unsigned short)(buf[5] & 0X3F) << 4) |
                                    (buf[4] & 0X0F);
170 }
171 
172 /*
173  * @description   : 打开设备
174  * @param – inode : 传递给驱动的 inode
175  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
176  *                  一般在 open 的时候将 private_data 指向设备结构体。
177  * @return        : 0 成功;其他 失败
178  */
179 static int ap3216c_open(struct inode *inode, struct file *filp)
180 {
181     filp->private_data = &ap3216cdev;
```

```
182 
183     /* 初始化 AP3216C */
184     ap3216c_write_reg(&ap3216cdev, AP3216C_SYSTEMCONG, 0x04);
185     mdelay(50);   /* AP3216C 复位最少 10ms  */
186     ap3216c_write_reg(&ap3216cdev, AP3216C_SYSTEMCONG, 0X03);
187     return 0;
188 }
189 
190 /*
191  * @description   : 从设备读取数据
192  * @param - filp  : 要打开的设备文件(文件描述符)
193  * @param - buf   : 返回给用户空间的数据缓冲区
194  * @param - cnt   : 要读取的数据长度
195  * @param - offt  : 相对于文件首地址的偏移
196  * @return        : 读取的字节数，如果为负值，表示读取失败
197  */
198 static ssize_t ap3216c_read(struct file *filp, char __user *buf,
                               size_t cnt, loff_t *off)
199 {
200     short data[3];
201     long err = 0;
202 
203     struct ap3216c_dev *dev = (struct ap3216c_dev *)
                                  filp->private_data;
204 
205     ap3216c_readdata(dev);
206 
207     data[0] = dev->ir;
208     data[1] = dev->als;
209     data[2] = dev->ps;
210     err = copy_to_user(buf, data, sizeof(data));
211     return 0;
212 }
213 
214 /*
215  * @description   : 关闭/释放设备
216  * @param - filp  : 要关闭的设备文件(文件描述符)
217  * @return        : 0 成功;其他 失败
218  */
219 static int ap3216c_release(struct inode *inode, struct file *filp)
220 {
221     return 0;
222 }
```

```
223
224 /* AP3216C 操作函数 */
225 static const struct file_operations ap3216c_ops = {
226     .owner = THIS_MODULE,
227     .open = ap3216c_open,
228     .read = ap3216c_read,
229     .release = ap3216c_release,
230 };
231
232  /*
233   * @description     : i2c 驱动的 probe 函数，当驱动与
234   *                    设备匹配以后此函数就会执行
235   * @param - client  : i2c 设备
236   * @param - id      : i2c 设备 ID
237   * @return          : 0，成功;其他负值,失败
238   */
239 static int ap3216c_probe(struct i2c_client *client,
                             const struct i2c_device_id *id)
240 {
241     /* 1、构建设备号 */
242     if (ap3216cdev.major) {
243         ap3216cdev.devid = MKDEV(ap3216cdev.major, 0);
244         register_chrdev_region(ap3216cdev.devid, AP3216C_CNT,
                                       AP3216C_NAME);
245     } else {
246         alloc_chrdev_region(&ap3216cdev.devid, 0, AP3216C_CNT,
                                     AP3216C_NAME);
247         ap3216cdev.major = MAJOR(ap3216cdev.devid);
248     }
249
250     /* 2、注册设备 */
251     cdev_init(&ap3216cdev.cdev, &ap3216c_ops);
252     cdev_add(&ap3216cdev.cdev, ap3216cdev.devid, AP3216C_CNT);
253
254     /* 3、创建类 */
255     ap3216cdev.class = class_create(THIS_MODULE, AP3216C_NAME);
256     if (IS_ERR(ap3216cdev.class)) {
257         return PTR_ERR(ap3216cdev.class);
258     }
259
260     /* 4、创建设备 */
261     ap3216cdev.device = device_create(ap3216cdev.class, NULL,
                                       ap3216cdev.devid, NULL, AP3216C_NAME);
```

```
262     if (IS_ERR(ap3216cdev.device)) {
263         return PTR_ERR(ap3216cdev.device);
264     }
265
266     ap3216cdev.private_data = client;
267
268     return 0;
269 }
270
271 /*
272  * @description     : i2c 驱动的 remove 函数，移除 i2c 驱动此函数会执行
273  * @param - client  : i2c 设备
274  * @return          : 0，成功;其他负值,失败
275  */
276 static int ap3216c_remove(struct i2c_client *client)
277 {
278     /* 删除设备 */
279     cdev_del(&ap3216cdev.cdev);
280     unregister_chrdev_region(ap3216cdev.devid, AP3216C_CNT);
281
282     /* 注销掉类和设备 */
283     device_destroy(ap3216cdev.class, ap3216cdev.devid);
284     class_destroy(ap3216cdev.class);
285     return 0;
286 }
287
288 /* 传统匹配方式 ID 列表 */
289 static const struct i2c_device_id ap3216c_id[] = {
290     {"alientek,ap3216c", 0},
291     {}
292 };
293
294 /* 设备树匹配列表 */
295 static const struct of_device_id ap3216c_of_match[] = {
296     { .compatible = "alientek,ap3216c" },
297     { /* Sentinel */ }
298 };
299
300 /* i2c 驱动结构体 */
301 static struct i2c_driver ap3216c_driver = {
302     .probe = ap3216c_probe,
303     .remove = ap3216c_remove,
304     .driver = {
```

```
305             .owner = THIS_MODULE,
306             .name = "ap3216c",
307             .of_match_table = ap3216c_of_match,
308         },
309     .id_table = ap3216c_id,
310 };
311
312 /*
313  * @description  : 驱动入口函数
314  * @param        : 无
315  * @return       : 无
316  */
317 static int __init ap3216c_init(void)
318 {
319     int ret = 0;
320
321     ret = i2c_add_driver(&ap3216c_driver);
322     return ret;
323 }
324
325 /*
326  * @description  : 驱动出口函数
327  * @param        : 无
328  * @return       : 无
329  */
330 static void __exit ap3216c_exit(void)
331 {
332     i2c_del_driver(&ap3216c_driver);
333 }
334
335 /* module_i2c_driver(ap3216c_driver) */
336
337 module_init(ap3216c_init);
338 module_exit(ap3216c_exit);
339 MODULE_LICENSE("GPL");
340 MODULE_AUTHOR("zuozhongkai");
```

第 32~41 行，ap3216c 设备结构体，第 39 行的 private_data 成员变量用于存放 ap3216c 对应的 i2c_client。第 40 行的 ir、als 和 ps 分别存储 AP3216C 的 IR、ALS 和 PS 数据。

第 43 行，定义一个 ap3216c_dev 类型的设备结构体变量 ap3216cdev。

第 53~79 行，ap3216c_read_regs 函数实现多字节读取，但是 AP3216C 好像不支持连续多字节读取，此函数在测试其他 I2C 设备的时候可以实现多给字节连续读取，但是在 AP3216C 上不能连续读取多个字节。不过读取一个字节没有问题的。

第 89~105 行，ap3216c_write_regs 函数实现连续多字节写操作。

第 113~124 行，ap3216c_read_reg 函数用于读取 AP3216C 的指定寄存器数据，用于一个寄存器的数据读取。

第 133~138 行，ap3216c_write_reg 函数用于向 AP3216C 的指定寄存器写入数据，用于一个寄存器的数据写操作。

第 148~170 行，读取 AP3216C 的 PS、ALS 和 IR 等传感器原始数据值。

第 179~230 行，标准的支付设备驱动框架。

第 239~269 行，ap3216c_probe 函数，当 I2C 设备和驱动匹配成功以后此函数就会执行，和 platform 驱动框架一样。此函数前面都是标准的字符设备注册代码，最后面会将此函数的第一个参数 client 传递给 ap3216cdev 的 private_data 成员变量。

第 289~292 行，ap3216c_id 匹配表，i2c_device_id 类型。用于传统的设备和驱动匹配，也就是没有使用设备树的时候。

第 295~298 行，ap3216c_of_match 匹配表，of_device_id 类型，用于设备树设备和驱动匹配。这里只写了一个 compatible 属性，值为“alientek,ap3216c”。

第 301~310 行，ap3216c_driver 结构体变量，i2c_driver 类型。

第 317~323 行，驱动入口函数 ap3216c_init，此函数通过调用 i2c_add_driver 来向 Linux 内核注册 i2c_driver，也就是 ap3216c_driver。

第 330~333 行，驱动出口函数 ap3216c_exit，此函数通过调用 i2c_del_driver 来注销掉前面注册的 ap3216c_driver。

61.5.3 编写测试 APP

新建 ap3216cApp.c 文件，然后在里面输入如下所示内容：

示例代码 61.5.3.1 ap3216cApp.c 文件代码段

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "sys/ioctl.h"
6  #include "fcntl.h"
7  #include "stdlib.h"
8  #include "string.h"
9  #include <poll.h>
10 #include <sys/select.h>
11 #include <sys/time.h>
12 #include <signal.h>
13 #include <fcntl.h>
14 /***************************************************************
15 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
16 文件名        : ap3216cApp.c
17 作者          : 左忠凯
18 版本          : V1.0
19 描述          : ap3216c 设备测试 APP。
20 其他          : 无
21 使用方法      : ./ap3216cApp /dev/ap3216c
```

```
22 论坛          : www.openedv.com
23 日志          : 初版 V1.0 2019/9/20 左忠凯创建
24 ***************************************************************/
25
26 /*
27  * @description    : main 主程序
28  * @param - argc   : argv 数组元素个数
29  * @param - argv   : 具体参数
30  * @return         : 0 成功;其他 失败
31  */
32 int main(int argc, char *argv[])
33 {
34     int fd;
35     char *filename;
36     unsigned short databuf[3];
37     unsigned short ir, als, ps;
38     int ret = 0;
39
40     if (argc != 2) {
41         printf("Error Usage!\r\n");
42         return -1;
43     }
44
45     filename = argv[1];
46     fd = open(filename, O_RDWR);
47     if(fd < 0) {
48         printf("can't open file %s\r\n", filename);
49         return -1;
50     }
51
52     while (1) {
53         ret = read(fd, databuf, sizeof(databuf));
54         if(ret == 0) {            /* 数据读取成功     */
55             ir =  databuf[0];     /* ir 传感器数据    */
56             als = databuf[1];     /* als 传感器数据   */
57             ps =  databuf[2];     /* ps 传感器数据    */
58             printf("ir = %d, als = %d, ps = %d\r\n", ir, als, ps);
59         }
60         usleep(200000);           /*100ms          */
61     }
62     close(fd);                    /* 关闭文件         */
63     return 0;
64 }
```

ap3216cApp.c 文件内容很简单，就是在 while 循环中不断的读取 AP3216C 的设备文件，从而得到 ir、als 和 ps 这三个数据值，然后将其输出到终端上。

61.6 运行测试

61.6.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为“ap3216c.o”，Makefile 内容如下所示：

示例代码 61.6.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := ap3216c.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“ap3216c.o”。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“ap3216c.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译 ap3216cApp.c 这个测试程序：

```
arm-linux-gnueabihf-gcc ap3216cApp.c -o ap3216cApp
```

编译成功以后就会生成 ap3216cApp 这个应用程序。

61.6.2 运行测试

将上一小节编译出来 ap3216c.ko 和 ap3216cApp 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中。输入如下命令加载 ap3216c.ko 这个驱动模块。

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe ap3216c.ko     //加载驱动模块
```

当驱动模块加载成功以后使用 ap3216cApp 来测试，输入如下命令：

```
./ap3216cApp /dev/ap3216c
```

测试 APP 会不断的从 AP3216C 中读取数据，然后输出到终端上，如图 61.6.2.1 所示：

```
/lib/modules/4.1.15 # ./ap3216cApp /dev/ap3216c
ir = 0, als = 0, ps = 0
ir = 0, als = 22, ps = 0
ir = 0, als = 18, ps = 0
ir = 4, als = 21, ps = 0
ir = 3, als = 20, ps = 0
ir = 0, als = 22, ps = 0
ir = 3, als = 19, ps = 0
ir = 0, als = 22, ps = 0
```

图 61.6.2.1 获取到的 AP3216C 数据

大家可以用手电筒照一下 AP3216C，或者手指靠近 AP3216C 来观察传感器数据有没有变化。

第六十二章 Linux SPI 驱动实验

上一章我们讲解了如何编写 Linux 下的 I2C 设备驱动，SPI 也是很常用的一个串行通信协议，本章我们就来学习一下如何在 Linux 下编写 SPI 设备驱动。本章实验的最终目的就是驱动 I.MX6U-ALPHA 开发板上的 ICM-20608 这个 SPI 接口的六轴传感器，可以在应用程序中读取 ICM-20608 的原始传感器数据。

62.1 Linux 下 SPI 驱动框架简介

SPI 驱动框架和 I2C 很类似，都分为主机控制器驱动和设备驱动，主机控制器也就是发 SOC 的 SPI 控制器接口。比如在裸机篇中的《第二十七章 SPI 实验》，我们编写了 bsp_spi.c 和 bsp_spi.h 这两个文件，这两个文件是 I.MX6U 的 SPI 控制器驱动，我们编写好 SPI 控制器驱动以后就可以直接使用了，不管是什么 SPI 设备，SPI 控制器部分的驱动都是一样，我们的重点就落在了种类繁多的 SPI 设备驱动。

62.1.1 SPI 主机驱动

SPI 主机驱动就是 SOC 的 SPI 控制器驱动，类似 I2C 驱动里面的适配器驱动。Linux 内核使用 spi_master 表示 SPI 主机驱动，spi_master 是个结构体，定义在 include/linux/spi/spi.h 文件中，内容如下(有缩减)：

示例代码 62.1.1.1 spi_master 结构体

```
315 struct spi_master {
316     struct device   dev;
317 
318     struct list_head list;
......
326     s16         bus_num;
327 
328     /* chipselects will be integral to many controllers; some others
329      * might use board-specific GPIOs.
330      */
331     u16         num_chipselect;
332 
333     /* some SPI controllers pose alignment requirements on DMAable
334      * buffers; let protocol drivers know about these requirements.
335      */
336     u16         dma_alignment;
337 
338     /* spi_device.mode flags understood by this controller driver */
339     u16         mode_bits;
340 
341     /* bitmask of supported bits_per_word for transfers */
342     u32         bits_per_word_mask;
......
347     /* limits on transfer speed */
348     u32         min_speed_hz;
349     u32         max_speed_hz;
350 
351     /* other constraints relevant to this driver */
352     u16         flags;
......
```

```
359     /* lock and mutex for SPI bus locking */
360     spinlock_t      bus_lock_spinlock;
361     struct mutex        bus_lock_mutex;
362 
363   /* flag indicating that the SPI bus is locked for exclusive use */
364     bool            bus_lock_flag;
......
372     int         (*setup)(struct spi_device *spi);
373 
......
393     int         (*transfer)(struct spi_device *spi,
394                         struct spi_message *mesg);
......
434     int (*transfer_one_message)(struct spi_master *master,
435                     struct spi_message *mesg);
......
462 };
```

第 393 行，transfer 函数，和 i2c_algorithm 中的 master_xfer 函数一样，控制器数据传输函数。

第 434 行，transfer_one_message 函数，也用于 SPI 数据发送，用于发送一个 spi_message，SPI 的数据会打包成 spi_message，然后以队列方式发送出去。

也就是 SPI 主机端最终会通过 transfer 函数与 SPI 设备进行通信，因此对于 SPI 主机控制器的驱动编写者而言 transfer 函数是需要实现的，因为不同的 SOC 其 SPI 控制器不同，寄存器都不一样。和 I2C 适配器驱动一样，SPI 主机驱动一般都是 SOC 厂商去编写的，所以我们作为 SOC 的使用者，这一部分的驱动就不用操心了，除非你是在 SOC 原厂工作，内容就是写 SPI 主机驱动。

SPI 主机驱动的核心就是申请 spi_master，然后初始化 spi_master，最后向 Linux 内核注册 spi_master。

1、spi_master 申请与释放

spi_alloc_master 函数用于申请 spi_master，函数原型如下：

```
struct spi_master *spi_alloc_master(struct device        *dev,
                                    unsigned             size)
```

函数参数和返回值含义如下：

dev：设备，一般是 platform_device 中的 dev 成员变量。

size：私有数据大小，可以通过 spi_master_get_devdata 函数获取到这些私有数据。

返回值：申请到的 spi_master。

spi_master 的释放通过 spi_master_put 函数来完成，当我们删除一个 SPI 主机驱动的时候就需要释放掉前面申请的 spi_master，spi_master_put 函数原型如下：

```
void spi_master_put(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要释放的 spi_master。

返回值：无。

2、spi_master 的注册与注销

当 spi_master 初始化完成以后就需要将其注册到 Linux 内核，spi_master 注册函数为 spi_register_master，函数原型如下：

```
int spi_register_master(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要注册的 spi_master。

返回值：0，成功；负值，失败。

I.MX6U 的 SPI 主机驱动会采用 spi_bitbang_start 这个 API 函数来完成 spi_master 的注册，spi_bitbang_start 函数内部其实也是通过调用 spi_register_master 函数来完成 spi_master 的注册。

如果要注销 spi_master 的话可以使用 spi_unregister_master 函数，此函数原型为：

```
void spi_unregister_master(struct spi_master *master)
```

函数参数和返回值含义如下：

master：要注销的 spi_master。

返回值：无。

如果使用 spi_bitbang_start 注册 spi_master 的话就要使用 spi_bitbang_stop 来注销掉 spi_master。

62.1.2 SPI 设备驱动

spi 设备驱动也和 i2c 设备驱动也很类似，Linux 内核使用 spi_driver 结构体来表示 spi 设备驱动，我们在编写 SPI 设备驱动的时候需要实现 spi_driver。spi_driver 结构体定义在 include/linux/spi/spi.h 文件中，结构体内容如下：

示例代码 62.1.1.2 spi_driver 结构体

```
180 struct spi_driver {
180     const struct spi_device_id *id_table;
180     int         (*probe)(struct spi_device *spi);
180     int         (*remove)(struct spi_device *spi);
180     void            (*shutdown)(struct spi_device *spi);
180     struct device_driver    driver;
180 };
```

可以看出，spi_driver 和 i2c_driver、platform_driver 基本一样，当 SPI 设备和驱动匹配成功以后 probe 函数就会执行。

同样的，spi_driver 初始化完成以后需要向 Linux 内核注册，spi_driver 注册函数为 spi_register_driver，函数原型如下：

```
int spi_register_driver(struct spi_driver *sdrv)
```

函数参数和返回值含义如下：

sdrv：要注册的 spi_driver。

返回值：0，注册成功；赋值，注册失败。

注销 SPI 设备驱动以后也需要注销掉前面注册的 spi_driver，使用 spi_unregister_driver 函数完成 spi_driver 的注销，函数原型如下：

```
void spi_unregister_driver(struct spi_driver *sdrv)
```

函数参数和返回值含义如下：

sdrv：要注销的 spi_driver。

返回值：无。

spi_driver 注册示例程序如下：

示例代码 62.1.1.3 spi_driver 注册示例程序

```
1  /* probe 函数 */
2  static int xxx_probe(struct spi_device *spi)
3  {
4      /* 具体函数内容 */
5      return 0;
6  }
7
8  /* remove 函数 */
9  static int xxx_remove(struct spi_device *spi)
10 {
11     /* 具体函数内容 */
12     return 0;
13 }
14 /* 传统匹配方式 ID 列表 */
15 static const struct spi_device_id xxx_id[] = {
16     {"xxx", 0},
17     {}
18 };
19
20 /* 设备树匹配列表 */
21 static const struct of_device_id xxx_of_match[] = {
22     { .compatible = "xxx" },
23     { /* Sentinel */ }
24 };
25
26 /* SPI 驱动结构体 */
27 static struct spi_driver xxx_driver = {
28     .probe = xxx_probe,
29     .remove = xxx_remove,
30     .driver = {
31         .owner = THIS_MODULE,
32         .name = "xxx",
33         .of_match_table = xxx_of_match,
34         },
35     .id_table = xxx_id,
36 };
37
38 /* 驱动入口函数 */
39 static int __init xxx_init(void)
40 {
```

```
41     return spi_register_driver(&xxx_driver);
42 }
43 
44 /* 驱动出口函数 */
45 static void __exit xxx_exit(void)
46 {
47     spi_unregister_driver(&xxx_driver);
48 }
49 
50 module_init(xxx_init);
51 module_exit(xxx_exit);
```

第 1~36 行，spi_driver 结构体，需要 SPI 设备驱动人员编写，包括匹配表、probe 函数等。和 i2c_driver、platform_driver 一样，就不详细讲解了。

第 39~42 行，在驱动入口函数中调用 spi_register_driver 来注册 spi_driver。

第 45~48 行，在驱动出口函数中调用 spi_unregister_driver 来注销 spi_driver。

62.1.3 SPI 设备和驱动匹配过程

SPI 设备和驱动的匹配过程是由 SPI 总线来完成的，这点和 platform、I2C 等驱动一样，SPI 总线为 spi_bus_type，定义在 drivers/spi/spi.c 文件中，内容如下：

示例代码 62.1.3.1 spi_bus_type 结构体

```
131 struct bus_type spi_bus_type = {
132     .name       = "spi",
133     .dev_groups = spi_dev_groups,
134     .match      = spi_match_device,
135     .uevent     = spi_uevent,
136 };
```

可以看出，SPI 设备和驱动的匹配函数为 spi_match_device，函数内容如下：

示例代码 62.1.3.2 spi_match_device 函数

```
99  static int spi_match_device(struct device *dev,
                                struct device_driver *drv)
100 {
101     const struct spi_device *spi = to_spi_device(dev);
102     const struct spi_driver *sdrv = to_spi_driver(drv);
103 
104     /* Attempt an OF style match */
105     if (of_driver_match_device(dev, drv))
106         return 1;
107 
108     /* Then try ACPI */
109     if (acpi_driver_match_device(dev, drv))
110         return 1;
111 
112     if (sdrv->id_table)
```

```
113         return !!spi_match_id(sdrv->id_table, spi);
114 
115     return strcmp(spi->modalias, drv->name) == 0;
116 }
```

spi_match_device 函数和 i2c_match_device 函数的对于设备和驱动的匹配过程基本一样。

第 105 行，of_driver_match_device 函数用于完成设备树设备和驱动匹配。比较 SPI 设备节点的 compatible 属性和 of_device_id 中的 compatible 属性是否相等，如果相当的话就表示 SPI 设备和驱动匹配。

第 109 行，acpi_driver_match_device 函数用于 ACPI 形式的匹配。

第 113 行，i2c_match_id 函数用于传统的、无设备树的 I2C 设备和驱动匹配过程。比较 I2C 设备名字和 i2c_device_id 的 name 字段是否相等，相等的话就说明 I2C 设备和驱动匹配。

第 115 行，比较 spi_device 中 modalias 成员变量和 device_driver 中的 name 成员变量是否相等。

62.2 I.MX6U SPI 主机驱动分析

和 I2C 的适配器驱动一样，SPI 主机驱动一般都由 SOC 厂商编写好了，打开 imx6ull.dtsi 文件，找到如下所示内容：

示例代码 62.2.1 imx6ull.dtsi 文件中的 ecspi3 节点内容

```
1  ecspi3: ecspi@02010000 {
2      #address-cells = <1>;
3      #size-cells = <0>;
4      compatible = "fsl,imx6ul-ecspi", "fsl,imx51-ecspi";
5      reg = <0x02010000 0x4000>;
6      interrupts = <GIC_SPI 33 IRQ_TYPE_LEVEL_HIGH>;
7      clocks = <&clks IMX6UL_CLK_ECSPI3>,
8          <&clks IMX6UL_CLK_ECSPI3>;
9      clock-names = "ipg", "per";
10     dmas = <&sdma 7 7 1>, <&sdma 8 7 2>;
11     dma-names = "rx", "tx";
12     status = "disabled";
13 };
```

重点来看一下第 4 行的 compatible 属性值，compatible 属性有两个值“fsl,imx6ul-ecspi”和“fsl,imx51-ecspi”，在 Linux 内核源码中搜素这两个属性值即可找到 I.MX6U 对应的 ECSPI(SPI)主机驱动。I.MX6U 的 ECSPI 主机驱动文件为 drivers/spi/spi-imx.c，在此文件中找到如下内容：

示例代码 62.2.2 spi_imx_driver 结构体

```
694 static struct platform_device_id spi_imx_devtype[] = {
695     {
696         .name = "imx1-cspi",
697         .driver_data = (kernel_ulong_t) &imx1_cspi_devtype_data,
698     }, {
699         .name = "imx21-cspi",
700         .driver_data = (kernel_ulong_t) &imx21_cspi_devtype_data,
......
```

```
713     }, {
714         .name = "imx6ul-ecspi",
715         .driver_data = (kernel_ulong_t) &imx6ul_ecspi_devtype_data,
716     }, {
717         /* sentinel */
718     }
719 };
720
721 static const struct of_device_id spi_imx_dt_ids[] = {
722     { .compatible = "fsl,imx1-cspi", .data =
                                &imx1_cspi_devtype_data, },
......
728     { .compatible = "fsl,imx6ul-ecspi", .data =
                                &imx6ul_ecspi_devtype_data, },
729     { /* sentinel */ }
730 };
731 MODULE_DEVICE_TABLE(of, spi_imx_dt_ids);
......
1338 static struct platform_driver spi_imx_driver = {
1339    .driver = {
1340            .name = DRIVER_NAME,
1341            .of_match_table = spi_imx_dt_ids,
1342            .pm = IMX_SPI_PM,
1343    },
1344    .id_table = spi_imx_devtype,
1345    .probe = spi_imx_probe,
1346    .remove = spi_imx_remove,
1347 };
1348 module_platform_driver(spi_imx_driver);
```

第 714 行，spi_imx_devtype 为 SPI 无设备树匹配表。

第 721 行，spi_imx_dt_ids 为 SPI 设备树匹配表。

第 728 行，“fsl,imx6ul-ecspi”匹配项，因此可知 I.MX6U 的 ECSPI 驱动就是 spi-imx.c 这个文件。

第 1338~1347 行，platform_driver 驱动框架，和 I2C 的适配器驱动一行，SPI 主机驱动器采用了 platfom 驱动框架。当设备和驱动匹配成功以后 spi_imx_probe 函数就会执行。

spi_imx_probe 函数会从设备树中读取相应的节点属性值，申请并初始化 spi_master，最后调用 spi_bitbang_start 函数(spi_bitbang_start 会调用 spi_register_master 函数)向 Linux 内核注册 spi_master。

对于 I.MX6U 来讲，SPI 主机的最终数据收发函数为 spi_imx_transfer，此函数通过如下层层调用最终实现 SPI 数据发送：

```
spi_imx_transfer
    -> spi_imx_pio_transfer
        -> spi_imx_push
```

```
-> spi_imx->tx
```

spi_imx 是个 spi_imx_data 类型的机构指针变量，其中 tx 和 rx 这两个成员变量分别为 SPI 数据发送和接收函数。I.MX6U SPI 主机驱动会维护一个 spi_imx_data 类型的变量 spi_imx，并且使用 spi_imx_setupxfer 函数来设置 spi_imx 的 tx 和 rx 函数。根据要发送的数据数据位宽的不同，分别有 8 位、16 位和 32 位的发送函数，如下所示：

```
spi_imx_buf_tx_u8
spi_imx_buf_tx_u16
spi_imx_buf_tx_u32
```

同理，也有 8 位、16 位和 32 位的数据接收函数，如下所示：

```
spi_imx_buf_rx_u8
spi_imx_buf_rx_u16
spi_imx_buf_rx_u32
```

我们就以 spi_imx_buf_tx_u8 这个函数为例，看看，一个自己的数据发送是怎么完成的，在 spi-imx.c 文件中找到如下所示内容：

示例代码 62.2.3 spi_imx_buf_tx_u8 函数

```
152 #define MXC_SPI_BUF_TX(type)                                    \
153 static void spi_imx_buf_tx_##type(struct spi_imx_data *spi_imx)  \
154 {                                                               \
155     type val = 0;                                               \
156                                                                 \
157     if (spi_imx->tx_buf) {                                      \
158         val = *(type *)spi_imx->tx_buf;                         \
159         spi_imx->tx_buf += sizeof(type);                        \
160     }                                                           \
161                                                                 \
162     spi_imx->count -= sizeof(type);                             \
163                                                                 \
164     writel(val, spi_imx->base + MXC_CSPITXDATA);                \
165 }
166
167 MXC_SPI_BUF_RX(u8)
168 MXC_SPI_BUF_TX(u8)
```

从示例代码 62.2.3 可以看出，spi_imx_buf_tx_u8 函数是通过 MXC_SPI_BUF_TX 宏来实现的。第 164 行就是将要发送的数据值写入到 ECSPI 的 TXDATA 寄存器里面去，这和我们 SPI 裸机实验的方法一样。将第 168 行的 MXC_SPI_BUF_TX(u8)展开就是 spi_imx_buf_tx_u8 函数。其他的 tx 和 rx 函数都是这样实现的，这里就不做介绍了。关于 I.MX6U 的主机驱动程序就讲解到这里，基本套路和 I2C 的适配器驱动程序类似。

62.3 SPI 设备驱动编写流程

62.3.1 SPI 设备信息描述

1、IO 的 pinctrl 子节点创建与修改

首先肯定是根据所使用的 IO 来创建或修改 pinctrl 子节点，这个没什么好说的，唯独要注

意的就是检查相应的 IO 有没有被其他的设备所使用，如果有的话需要将其删除掉！

2、SPI 设备节点的创建与修改

采用设备树的情况下，SPI 设备信息描述就通过创建相应的设备子节点来完成，我们可以打开 imx6qdl-sabresd.dtsi 这个设备树头文件，在此文件里面找到如下所示内容：

示例代码 62.3.1.1 m25p80 设备节点

```
308 &ecspi1 {
309     fsl,spi-num-chipselects = <1>;
310     cs-gpios = <&gpio4 9 0>;
311     pinctrl-names = "default";
312     pinctrl-0 = <&pinctrl_ecspi1>;
313     status = "okay";
314 
315     flash: m25p80@0 {
316         #address-cells = <1>;
317         #size-cells = <1>;
318         compatible = "st,m25p32";
319         spi-max-frequency = <20000000>;
320         reg = <0>;
321     };
322 };
```

示例代码 62.3.1.1 是 I.MX6Q 的一款板子上的一个 SPI 设备节点，在这个板子的 ECSPI 接口上接了一个 m25p80，这是一个 SPI 接口的设备。

第 309 行，设置“fsl,spi-num-chipselects”属性为 1，表示只有一个设备。

第 310 行，设置“cs-gpios”属性，也就是片选信号为 GPIO4_IO09。

第 311 行，设置“pinctrl-names”属性，也就是 SPI 设备所使用的 IO 名字。

第 312 行，设置“pinctrl-0”属性，也就是所使用的 IO 对应的 pinctrl 节点。

第 313 行，将 ecspi1 节点的“status”属性改为“okay”。

第 315~320 行，ecspi1 下的 m25p80 设备信息，每一个 SPI 设备都采用一个子节点来描述其设备信息。第 315 行的“m25p80@0”后面的“0”表示 m25p80 的接到了 ECSPI 的通道 0 上。这个要根据自己的具体硬件来设置。

第 318 行，SPI 设备的 compatible 属性值，用于匹配设备驱动。

第 319 行，“spi-max-frequency”属性设置 SPI 控制器的最高频率，这个要根据所使用的 SPI 设备来设置，比如在这里将 SPI 控制器最高频率设置为 20MHz。

第 320 行，reg 属性设置 m25p80 这个设备所使用的 ECSPI 通道，和“m25p80@0”后面的“0”一样。

我们一会在编写 ICM20608 的设备树节点信息的时候就参考示例代码 62.3.1.1 中的内容即可。

62.3.2 SPI 设备数据收发处理流程

SPI 设备驱动的核心是 spi_driver，这个我们已经在 62.1.2 小节讲过了。当我们向 Linux 内核注册成功 spi_driver 以后就可以使用 SPI 核心层提供的 API 函数来对设备进行读写操作了。首先是 spi_transfer 结构体，此结构体用于描述 SPI 传输信息，结构体内容如下：

示例代码 62.3.2.1 spi_transfer 结构体

```
603 struct spi_transfer {
604     /* it's ok if tx_buf == rx_buf (right?)
605      * for MicroWire, one buffer must be null
606      * buffers must work with dma_*map_single() calls, unless
607      *   spi_message.is_dma_mapped reports a pre-existing mapping
608      */
609     const void  *tx_buf;
610     void        *rx_buf;
611     unsigned    len;
612 
613     dma_addr_t  tx_dma;
614     dma_addr_t  rx_dma;
615     struct sg_table tx_sg;
616     struct sg_table rx_sg;
617 
618     unsigned    cs_change:1;
619     unsigned    tx_nbits:3;
620     unsigned    rx_nbits:3;
621 #define SPI_NBITS_SINGLE    0x01 /* 1bit transfer */
622 #define SPI_NBITS_DUAL      0x02 /* 2bits transfer */
623 #define SPI_NBITS_QUAD      0x04 /* 4bits transfer */
624     u8      bits_per_word;
625     u16     delay_usecs;
626     u32     speed_hz;
627 
628     struct list_head transfer_list;
629 };
```

第 609 行，tx_buf 保存着要发送的数据。

第 610 行，rx_buf 用于保存接收到的数据。

第 611 行，len 是要进行传输的数据长度，SPI 是全双工通信，因此在一次通信中发送和接收的字节数都是一样的，所以 spi_transfer 中也就没有发送长度和接收长度之分。

spi_transfer 需要组织成 spi_message，spi_message 也是一个结构体，内容如下：

示例代码 62.3.2.2 spi_message 结构体

```
660 struct spi_message {
661     struct list_head    transfers;
662 
663     struct spi_device   *spi;
664 
665     unsigned        is_dma_mapped:1;
......
678     /* completion is reported through a callback */
679     void            (*complete)(void *context);
```

```
680     void            *context;
681     unsigned        frame_length;
682     unsigned        actual_length;
683     int         status;
684 
685     /* for optional use by whatever driver currently owns the
686      * spi_message ...  between calls to spi_async and then later
687      * complete(), that's the spi_master controller driver.
688      */
689     struct list_head    queue;
690     void            *state;
691 };
```

在使用 spi_message 之前需要对其进行初始化，spi_message 初始化函数为 spi_message_init，函数原型如下：

```
void spi_message_init(struct spi_message *m)
```

函数参数和返回值含义如下：

m：要初始化的 spi_message。

返回值：无。

spi_message 初始化完成以后需要将 spi_transfer 添加到 spi_message 队列中，这里我们要用到 spi_message_add_tail 函数，此函数原型如下：

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
```

函数参数和返回值含义如下：

t：要添加到队列中的 spi_transfer。

m：spi_transfer 要加入的 spi_message。

返回值：无。

spi_message 准备好以后既可以进行数据传输了，数据传输分为同步传输和异步传输，同步传输会阻塞的等待 SPI 数据传输完成，同步传输函数为 spi_sync，函数原型如下：

```
int spi_sync(struct spi_device *spi, struct spi_message *message)
```

函数参数和返回值含义如下：

spi：要进行数据传输的 spi_device。

message：要传输的 spi_message。

返回值：无。

异步传输不会阻塞的等到 SPI 数据传输完成，异步传输需要设置 spi_message 中的 complete 成员变量，complete 是一个回调函数，当 SPI 异步传输完成以后此函数就会被调用。SPI 异步传输函数为 spi_async，函数原型如下：

```
int spi_async(struct spi_device *spi, struct spi_message *message)
```

函数参数和返回值含义如下：

spi：要进行数据传输的 spi_device。

message：要传输的 spi_message。

返回值：无。

在本章实验中，我们采用同步传输方式来完成 SPI 数据的传输工作，也就是 spi_sync 函数。

综上所述，SPI 数据传输步骤如下：

①、申请并初始化 spi_transfer，设置 spi_transfer 的 tx_buf 成员变量，tx_buf 为要发送的数据。然后设置 rx_buf 成员变量，rx_buf 保存着接收到的数据。最后设置 len 成员变量，也就是要进行数据通信的长度。

②、使用 spi_message_init 函数初始化 spi_message。

③、使用 spi_message_add_tail 函数将前面设置好的 spi_transfer 添加到 spi_message 队列中。

④、使用 spi_sync 函数完成 SPI 数据同步传输。

通过 SPI 进行 n 个字节的数据发送和接收的示例代码如下所示：

示例代码 62.3.2.3 SPI 数据读写操作

```
/* SPI 多字节发送 */
static int spi_send(struct spi_device *spi, u8 *buf, int len)
{
    int ret;
    struct spi_message m;

    struct spi_transfer t = {
        .tx_buf = buf,
        .len = len,
    };

    spi_message_init(&m);          /* 初始化 spi_message */
    spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message 队列 */
    ret = spi_sync(spi, &m);      /* 同步传输 */
    return ret;
}

/* SPI 多字节接收 */
static int spi_receive(struct spi_device *spi, u8 *buf, int len)
{
    int ret;
    struct spi_message m;

    struct spi_transfer t = {
        .rx_buf = buf,
        .len = len,
    };

    spi_message_init(&m);          /* 初始化 spi_message */
    spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message 队列 */
    ret = spi_sync(spi, &m);      /* 同步传输 */
    return ret;
}
```

62.4 硬件原理图分析

本章实验硬件原理图参考 26.2 小节即可。

62.5 试验程序编写

本实验对应的例程路径为：开发板光盘-> 2、Linux 驱动例程-> 22_spi。

62.5.1 修改设备树

1、添加 ICM20608 所使用的 IO

首先在 imx6ull-alientek-emmc.dts 文件中添加 ICM20608 所使用的 IO 信息，在 iomuxc 节点中添加一个新的子节点来描述 ICM20608 所使用的 SPI 引脚，子节点名字为 pinctrl_ecspi3，节点内容如下所示：

示例代码 62.5.1.1 icm20608 IO 节点信息

```
1 pinctrl_ecspi3: icm20608 {
2         fsl,pins = <
3             MX6UL_PAD_UART2_TX_DATA__GPIO1_IO20      0x10b0 /* CS */
4             MX6UL_PAD_UART2_RX_DATA__ECSPI3_SCLK     0x10b1 /* SCLK */
5             MX6UL_PAD_UART2_RTS_B__ECSPI3_MISO       0x10b1 /* MISO */
6             MX6UL_PAD_UART2_CTS_B__ECSPI3_MOSI       0x10b1 /* MOSI */
7         >;
8     };
```

UART2_TX_DATA 这个 IO 是 ICM20608 的片选信号，这里我们并没有将其复用为 ECSPI3 的 SS0 信号，而是将其复用为了普通的 GPIO。因为我们需要自己控制片选信号，所以将其复用为普通的 GPIO。

2、在 ecspi3 节点追加 icm20608 子节点

在 imx6ull-alientek-emmc.dts 文件中并没有任何向 ecspi3 节点追加内容的代码，这是因为 NXP 官方的 6ULL EVK 开发板上没有连接 SPI 设备。在 imx6ull-alientek-emmc.dts 文件最后面加入如下所示内容：

示例代码 62.5.1.2 向 ecspi3 节点加入 icm20608 信息

```
1  &ecspi3 {
2      fsl,spi-num-chipselects = <1>;
3      cs-gpio = <&gpio1 20 GPIO_ACTIVE_LOW>; /* cant't use cs-gpios! */
4      pinctrl-names = "default";
5      pinctrl-0 = <&pinctrl_ecspi3>;
6      status = "okay";
7
8      spidev: icm20608@0 {
9          compatible = "alientek,icm20608";
10         spi-max-frequency = <8000000>;
11         reg = <0>;
12     };
13 };
```

第 2 行，设置当前片选数量为 1，因为就只接了一个 ICM20608。

第 3 行，注意！这里并没有用到“cs-gpios”属性，而是用了一个自己定义的“cs-gpio”属性，因为我们要自己控制片选引脚。如果使用“cs-gpios”属性的话 SPI 主机驱动就会控制片选引脚。

第 5 行，设置 IO 要使用的 pinctrl 子节点，也就是我们在示例代码 62.5.1.1 中新建的 pinctrl_ecspi3。

第 6 行，imx6ull.dtsi 文件中默认将 ecspi3 节点状态(status)设置为“disable”，这里我们要将其改为“okay”。

第 8~12 行，icm20608 设备子节点，因为 icm20608 连接在 ECSPI3 的第 0 个通道上，因此@后面为 0。第 9 行设置节点属性兼容值为“alientek,icm20608”，第 10 行设置 SPI 最大时钟频率为 8MHz，这是 ICM20608 的 SPI 接口所能支持的最大的时钟频率。第 11 行，icm20608 连接在通道 0 上，因此 reg 为 0。

imx6ull-alientek-emmc.dts 文件修改完成以后重新编译一下，得到新的 dtb 文件，并使用新的 dtb 启动 Linux 系统。

62.5.2 编写 ICM20608 驱动

新建名为“22_spi”的文件夹，然后在 22_spi 文件夹里面创建 vscode 工程，工作区命名为“spi”。工程创建好以后新建 icm20608.c 和 icm20608reg.h 这两个文件，icm20608.c 为 ICM20608 的驱动代码，icm20608reg.h 是 ICM20608 寄存器头文件。先在 icm20608reg.h 中定义好 ICM20608 的寄存器，输入如下内容(有省略，完成的内容请参考例程)：

示例代码 62.5.2.1 icm20608reg.h 文件内容

```
1  #ifndef ICM20608_H
2  #define ICM20608_H
3  /***************************************************************
4  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
5  文件名         : icm20608reg.h
6  作者           : 左忠凯
7  版本           : V1.0
8  描述           : ICM20608 寄存器地址描述头文件
9  其他           : 无
10 论坛           : www.openedv.com
11 日志           : 初版 V1.0 2019/9/2 左忠凯创建
12 ***************************************************************/
13 #define ICM20608G_ID                    0XAF    /* ID 值 */
14 #define ICM20608D_ID                    0XAE    /* ID 值 */
15
16 /* ICM20608 寄存器
17  *复位后所有寄存器地址都为 0，除了
18  *Register 107(0X6B) Power Management 1    = 0x40
19  *Register 117(0X75) WHO_AM_I              = 0xAF 或 0xAE
20  */
21 /* 陀螺仪和加速度自测(出产时设置，用于与用户的自检输出值比较） */
22 #define  ICM20_SELF_TEST_X_GYRO      0x00
```

```
23 #define  ICM20_SELF_TEST_Y_GYRO      0x01
24 #define  ICM20_SELF_TEST_Z_GYRO      0x02
25 #define  ICM20_SELF_TEST_X_ACCEL     0x0D
26 #define  ICM20_SELF_TEST_Y_ACCEL     0x0E
27 #define  ICM20_SELF_TEST_Z_ACCEL     0x0F
......
80 /* 加速度静态偏移 */
81 #define  ICM20_XA_OFFSET_H           0x77
82 #define  ICM20_XA_OFFSET_L           0x78
83 #define  ICM20_YA_OFFSET_H           0x7A
84 #define  ICM20_YA_OFFSET_L           0x7B
85 #define  ICM20_ZA_OFFSET_H           0x7D
86 #define  ICM20_ZA_OFFSET_L           0x7E
87 
88 #endif
```

接下来继续编写 icm20608.c 文件，因为 icm20608.c 文件内容比较长，因此这里就将其分开来讲解。

1、icm20608 设备结构体创建

首先创建一个 icm20608 设备机构体，如下所示：

示例代码 62.5.2.2 icm20608 设备结构体创建

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
......
22 #include <asm/io.h>
23 #include "icm20608reg.h"
24 /***************************************************************
25 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
26 文件名       : icm20608.c
27 作者         : 左忠凯
28 版本         : V1.0
29 描述         : ICM20608 SPI 驱动程序
30 其他         : 无
31 论坛         : www.openedv.com
32 日志         : 初版 V1.0 2019/9/2 左忠凯创建
33 ***************************************************************/
34 #define ICM20608_CNT 1
35 #define ICM20608_NAME    "icm20608"
36 
37 struct icm20608_dev {
38     dev_t devid;                 /* 设备号             */
39     struct cdev cdev;            /* cdev               */
40     struct class *class;         /* 类                 */
```

```
41      struct device *device;         /* 设备                */
42      struct device_node  *nd;       /* 设备节点            */
43      int major;                     /* 主设备号            */
44      void *private_data;            /* 私有数据            */
45      int cs_gpio;                   /* 片选所使用的 GPIO 编号*/
46      signed int gyro_x_adc;         /* 陀螺仪 X 轴原始值    */
47      signed int gyro_y_adc;         /* 陀螺仪 Y 轴原始值    */
48      signed int gyro_z_adc;         /* 陀螺仪 Z 轴原始值    */
49      signed int accel_x_adc;        /* 加速度计 X 轴原始值  */
50      signed int accel_y_adc;        /* 加速度计 Y 轴原始值  */
51      signed int accel_z_adc;        /* 加速度计 Z 轴原始值  */
52      signed int temp_adc;           /* 温度原始值          */
53 };
54 
55 static struct icm20608_dev icm20608dev;
```

icm20608 的设备结构体 icm20608_dev 没什么好讲的，重点看一下第 44 行的 private_data，对于 SPI 设备驱动来讲最核心的就是 spi_device。probe 函数会向驱动提供当前 SPI 设备对应的 spi_device，因此在 probe 函数中设置 private_data 为 probe 函数传递进来的 spi_device 参数。

2、icm20608 的 spi_driver 注册与注销

对于 SPI 设备驱动，首先就是要初始化并向系统注册 spi_driver，icm20608 的 spi_driver 初始化、注册与注销代码如下：

示例代码 62.5.2.3 icm20608 的 spi_driver 初始化、注册与注销

```
1  /* 传统匹配方式 ID 列表 */
2  static const struct spi_device_id icm20608_id[] = {
3      {"alientek,icm20608", 0},
4      {}
5  };
6  
7  /* 设备树匹配列表 */
8  static const struct of_device_id icm20608_of_match[] = {
9      { .compatible = "alientek,icm20608" },
10     { /* Sentinel */ }
11 };
12 
13 /* SPI 驱动结构体 */
14 static struct spi_driver icm20608_driver = {
15     .probe = icm20608_probe,
16     .remove = icm20608_remove,
17     .driver = {
18         .owner = THIS_MODULE,
19         .name = "icm20608",
20         .of_match_table = icm20608_of_match,
21        },
```

```
22     .id_table = icm20608_id,
23 };
24
25 /*
26  * @description    : 驱动入口函数
27  * @param          : 无
28  * @return         : 无
29  */
30 static int __init icm20608_init(void)
31 {
32     return spi_register_driver(&icm20608_driver);
33 }
34
35 /*
36  * @description    : 驱动出口函数
37  * @param          : 无
38  * @return         : 无
39  */
40 static void __exit icm20608_exit(void)
41 {
42     spi_unregister_driver(&icm20608_driver);
43 }
44
45 module_init(icm20608_init);
46 module_exit(icm20608_exit);
47 MODULE_LICENSE("GPL");
48 MODULE_AUTHOR("zuozhongkai");
```

第 2~5 行，传统的设备和驱动匹配表。

第 8~11 行，设备树的设备与驱动匹配表，这里只有一个匹配项：“alientek,icm20608”。

第 14~23 行，icm20608 的 spi_driver 结构体变量，当 icm20608 设备和此驱动匹配成功以后第 15 行的 icm20608_probe 函数就会执行。同样的，当注销此驱动的时候 icm20608_remove 函数会执行。

第 30~33 行，icm20608_init 函数为 icm20608 的驱动入口函数，在此函数中使用 spi_register_driver 向 Linux 系统注册上面定义的 icm20608_driver。

第 40~43 行，icm20608_exit 函数为 icm20608 的驱动出口函数，在此函数中使用 spi_unregister_driver 注销掉前面注册的 icm20608_driver。

3、probe/remove 函数

icm20608_driver 中的 probe 和 remove 函数内容如下所示：

示例代码 62.5.2.4 probe 和 remove 函数

```
1  /*
2   * @description       : spi 驱动的 probe 函数，当驱动与
3   *                       设备匹配以后此函数就会执行
4   * @param - client    : spi 设备
```

```
5    * @param - id       : spi 设备 ID
6    *
7    */
8  static int icm20608_probe(struct spi_device *spi)
9  {
10     int ret = 0;
11 
12     /* 1、构建设备号 */
13     if (icm20608dev.major) {
14         icm20608dev.devid = MKDEV(icm20608dev.major, 0);
15         register_chrdev_region(icm20608dev.devid, ICM20608_CNT,
                                 ICM20608_NAME);
16     } else {
17         alloc_chrdev_region(&icm20608dev.devid, 0, ICM20608_CNT,
                                 ICM20608_NAME);
18         icm20608dev.major = MAJOR(icm20608dev.devid);
19     }
20 
21     /* 2、注册设备 */
22     cdev_init(&icm20608dev.cdev, &icm20608_ops);
23     cdev_add(&icm20608dev.cdev, icm20608dev.devid, ICM20608_CNT);
24 
25     /* 3、创建类 */
26     icm20608dev.class = class_create(THIS_MODULE, ICM20608_NAME);
27     if (IS_ERR(icm20608dev.class)) {
28         return PTR_ERR(icm20608dev.class);
29     }
30 
31     /* 4、创建设备 */
32     icm20608dev.device = device_create(icm20608dev.class, NULL,
                        icm20608dev.devid, NULL, ICM20608_NAME);
33     if (IS_ERR(icm20608dev.device)) {
34         return PTR_ERR(icm20608dev.device);
35     }
36 
37     /* 获取设备树中 cs 片选信号 */
38     icm20608dev.nd = of_find_node_by_path("/soc/aips-bus@02000000/
                            spba-bus@02000000/ecspi@02010000");
39     if(icm20608dev.nd == NULL) {
40         printk("ecspi3 node not find!\r\n");
41         return -EINVAL;
42     }
43 
```

```
44      /* 2、 获取设备树中的 gpio 属性，得到 BEEP 所使用的 BEEP 编号 */
45      icm20608dev.cs_gpio = of_get_named_gpio(icm20608dev.nd,
                                                  "cs-gpio", 0);
46      if(icm20608dev.cs_gpio < 0) {
47          printk("can't get cs-gpio");
48          return -EINVAL;
49      }
50
51      /* 3、设置 GPIO1_IO20 为输出，并且输出高电平 */
52      ret = gpio_direction_output(icm20608dev.cs_gpio, 1);
53      if(ret < 0) {
54          printk("can't set gpio!\r\n");
55      }
56
57      /*初始化 spi_device */
58      spi->mode = SPI_MODE_0;         /*MODE0，CPOL=0，CPHA=0   */
59      spi_setup(spi);
60      icm20608dev.private_data = spi; /* 设置私有数据            */
61
62      /* 初始化 ICM20608 内部寄存器 */
63      icm20608_reginit();
64      return 0;
65  }
66
67  /*
68   * @description    : spi 驱动的 remove 函数，移除 spi 驱动的时候此函数会执行
69   * @param – client : spi 设备
70   * @return         : 0，成功;其他负值,失败
71   */
72  static int icm20608_remove(struct spi_device *spi)
73  {
74      /* 删除设备 */
75      cdev_del(&icm20608dev.cdev);
76      unregister_chrdev_region(icm20608dev.devid, ICM20608_CNT);
77
78      /* 注销掉类和设备 */
79      device_destroy(icm20608dev.class, icm20608dev.devid);
80      class_destroy(icm20608dev.class);
81      return 0;
82  }
```

第 8~65 行，probe 函数，当设备与驱动匹配成功以后此函数就会执行，第 13~55 行都是标准的注册字符设备驱动。其中在第 38~49 行获取设备节点中的“cs-gpio”属性，也就是获取到设备的片选 IO。

第 58 行，设置 SPI 为模式 0，也就是 CPOL=0，CPHA=0。

第 59 行，设置好 spi_device 以后需要使用 spi_setup 配置一下。

第 60 行，设置 icm20608dev 的 private_data 成员变量为 spi_device。

第 63 行，调用 icm20608_reginit 函数初始化 ICM20608，主要是初始化 ICM20608 指定寄存器。

第 72~81 行，icm20608_remove 函数，注销驱动的时候此函数就会执行。

4、icm20608 寄存器读写与初始化

SPI 驱动的最终目的就是为了读写 icm20608 的寄存器，因此需要编写相应的寄存器读写函数，并且使用这些读写函数来完成对 icm20608 的初始化。icm20608 的寄存器读写以及初始化代码如下：

示例代码 62.5.2.5 icm20608 寄存器读写以及出初始化

```
1  /*
2   * @description   : 从 icm20608 读取多个寄存器数据
3   * @param - dev   :  icm20608 设备
4   * @param - reg   :  要读取的寄存器首地址
5   * @param - val   :  读取到的数据
6   * @param - len   :  要读取的数据长度
7   * @return        : 操作结果
8   */
9  static int icm20608_read_regs(struct icm20608_dev *dev, u8 reg,
                                  void *buf, int len)
10 {
11     int ret;
12     unsigned char txdata[len];
13     struct spi_message m;
14     struct spi_transfer *t;
15     struct spi_device *spi = (struct spi_device *)dev->private_data;
16 
17     gpio_set_value(dev->cs_gpio, 0);  /* 片选拉低，选中 ICM20608 */
18     t = kzalloc(sizeof(struct spi_transfer), GFP_KERNEL);
19 
20     /* 第 1 次，发送要读取的寄存地址 */
21     txdata[0] = reg | 0x80;      /* 写数据的时候寄存器地址 bit8 要置 1 */
22     t->tx_buf = txdata;          /* 要发送的数据                     */
23     t->len = 1;                  /* 1 个字节                         */
24     spi_message_init(&m);        /* 初始化 spi_message               */
25     spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message */
26     ret = spi_sync(spi, &m);     /* 同步发送                         */
27 
28     /* 第 2 次，读取数据 */
29     txdata[0] = 0xff;            /* 随便一个值，此处无意义           */
30     t->rx_buf = buf;             /* 读取到的数据                     */
31     t->len = len;                /* 要读取的数据长度                 */
```

```
32      spi_message_init(&m);       /* 初始化 spi_message            */
33      spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message*/
34      ret = spi_sync(spi, &m);    /* 同步发送                      */
35  
36      kfree(t);                          /* 释放内存                  */
37      gpio_set_value(dev->cs_gpio, 1);   /* 片选拉高，释放 ICM20608   */
38  
39      return ret;
40  }
41  
42  /*
43   * @description   : 向 icm20608 多个寄存器写入数据
44   * @param - dev   : icm20608 设备
45   * @param - reg   : 要写入的寄存器首地址
46   * @param - val   : 要写入的数据缓冲区
47   * @param - len   : 要写入的数据长度
48   * @return        : 操作结果
49   */
50  static s32 icm20608_write_regs(struct icm20608_dev *dev, u8 reg,
                                   u8 *buf, u8 len)
51  {
52      int ret;
53  
54      unsigned char txdata[len];
55      struct spi_message m;
56      struct spi_transfer *t;
57      struct spi_device *spi = (struct spi_device *)dev->private_data;
58  
59      t = kzalloc(sizeof(struct spi_transfer), GFP_KERNEL);
60      gpio_set_value(dev->cs_gpio, 0);   /* 片选拉低 */
61  
62      /* 第 1 次，发送要读取的寄存地址 */
63      txdata[0] = reg & ~0x80;    /* 写数据的时候寄存器地址 bit8 要清零 */
64      t->tx_buf = txdata;         /* 要发送的数据                  */
65      t->len = 1;                 /* 1 个字节                      */
66      spi_message_init(&m);       /* 初始化 spi_message            */
67      spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message */
68      ret = spi_sync(spi, &m);    /* 同步发送                      */
69  
70      /* 第 2 次，发送要写入的数据 */
71      t->tx_buf = buf;            /* 要写入的数据                  */
72      t->len = len;               /* 写入的字节数                  */
73      spi_message_init(&m);       /* 初始化 spi_message            */
```

```
74      spi_message_add_tail(t, &m);/* 将 spi_transfer 添加到 spi_message*/
75      ret = spi_sync(spi, &m);     /* 同步发送                      */
76
77      kfree(t);                    /* 释放内存                      */
78      gpio_set_value(dev->cs_gpio, 1);/* 片选拉高，释放 ICM20608     */
79      return ret;
80  }
81
82  /*
83   * @description   : 读取 icm20608 指定寄存器值，读取一个寄存器
84   * @param - dev   : icm20608 设备
85   * @param - reg   : 要读取的寄存器
86   * @return        : 读取到的寄存器值
87   */
88  static unsigned char icm20608_read_onereg(struct icm20608_dev *dev,
                                               u8 reg)
89  {
90      u8 data = 0;
91      icm20608_read_regs(dev, reg, &data, 1);
92      return data;
93  }
94
95  /*
96   * @description   : 向 icm20608 指定寄存器写入指定的值，写一个寄存器
97   * @param - dev   : icm20608 设备
98   * @param - reg   : 要写的寄存器
99   * @param - data  : 要写入的值
100  * @return        : 无
101  */
102
103 static void icm20608_write_onereg(struct icm20608_dev *dev, u8 reg,
                                      u8 value)
104 {
105     u8 buf = value;
106     icm20608_write_regs(dev, reg, &buf, 1);
107 }
108
109 /*
110  * @description   : 读取 ICM20608 的数据，读取原始数据，包括三轴陀螺仪、
111  *                : 三轴加速度计和内部温度。
112  * @param - dev   : ICM20608 设备
113  * @return        : 无。
114  */
```

```
115 void icm20608_readdata(struct icm20608_dev *dev)
116 {
117     unsigned char data[14];
118     icm20608_read_regs(dev, ICM20_ACCEL_XOUT_H, data, 14);
119 
120     dev->accel_x_adc = (signed short)((data[0] << 8) | data[1]);
121     dev->accel_y_adc = (signed short)((data[2] << 8) | data[3]);
122     dev->accel_z_adc = (signed short)((data[4] << 8) | data[5]);
123     dev->temp_adc    = (signed short)((data[6] << 8) | data[7]);
124     dev->gyro_x_adc  = (signed short)((data[8] << 8) | data[9]);
125     dev->gyro_y_adc  = (signed short)((data[10] << 8) | data[11]);
126     dev->gyro_z_adc  = (signed short)((data[12] << 8) | data[13]);
127 }
128 /*
129  * ICM20608 内部寄存器初始化函数
130  * @param      : 无
131  * @return     : 无
132  */
133 void icm20608_reginit(void)
134 {
135     u8 value = 0;
136 
137     icm20608_write_onereg(&icm20608dev, ICM20_PWR_MGMT_1, 0x80);
138     mdelay(50);
139     icm20608_write_onereg(&icm20608dev, ICM20_PWR_MGMT_1, 0x01);
140     mdelay(50);
141 
142     value = icm20608_read_onereg(&icm20608dev, ICM20_WHO_AM_I);
143     printk("ICM20608 ID = %#X\r\n", value);
144 
145     icm20608_write_onereg(&icm20608dev, ICM20_SMPLRT_DIV, 0x00);
146     icm20608_write_onereg(&icm20608dev, ICM20_GYRO_CONFIG, 0x18);
147     icm20608_write_onereg(&icm20608dev, ICM20_ACCEL_CONFIG, 0x18);
148     icm20608_write_onereg(&icm20608dev, ICM20_CONFIG, 0x04);
149     icm20608_write_onereg(&icm20608dev, ICM20_ACCEL_CONFIG2, 0x04);
150     icm20608_write_onereg(&icm20608dev, ICM20_PWR_MGMT_2, 0x00);
151     icm20608_write_onereg(&icm20608dev, ICM20_LP_MODE_CFG, 0x00);
152     icm20608_write_onereg(&icm20608dev, ICM20_FIFO_EN, 0x00);
153 }
```

第 9~40 行，icm20608_read_regs 函数，从 icm20608 中读取连续多个寄存器数据。

第 50~80 行，icm20608_write_regs 函数，向 icm20608 连续写入多个寄存器数据。

第 88~83 行，icm20608_read_onereg 函数，读取 icm20608 指定寄存器数据。

第 103~107 行，icm20608_write_onereg 函数，向 icm20608 指定寄存器写入数据。

第 115~126 行，icm20608_readdata 函数，读取 icm20608 六轴传感器和温度传感器原始数据值，应用程序读取 icm20608 的时候这些传感器原始数据就会上报给应用程序。

第 133~153 行，icm20608_reginit 函数，初始化 icm20608，和我们 spi 裸机实验里面的初始化过程一样。

5、字符设备驱动框架

icm20608 的字符设备驱动框架如下：

示例代码 62.5.2.6 icm20608 字符设备驱动

```
1  /*
2   * @description    : 打开设备
3   * @param – inode  : 传递给驱动的 inode
4   * @param - filp   : 设备文件，file 结构体有个叫做 pr 似有 ate_data 的成员变量
5   *                   一般在 open 的时候将 private_data 似有向设备结构体。
6   * @return         : 0 成功;其他 失败
7   */
8  static int icm20608_open(struct inode *inode, struct file *filp)
9  {
10     filp->private_data = &icm20608dev; /* 设置私有数据 */
11     return 0;
12 }
13
14 /*
15  * @description    : 从设备读取数据
16  * @param - filp   : 要打开的设备文件(文件描述符)
17  * @param - buf    : 返回给用户空间的数据缓冲区
18  * @param - cnt    : 要读取的数据长度
19  * @param - offt   : 相对于文件首地址的偏移
20  * @return         : 读取的字节数，如果为负值，表示读取失败
21  */
22 static ssize_t icm20608_read(struct file *filp, char __user *buf,
                                 size_t cnt, loff_t *off)
23 {
24     signed int data[7];
25     long err = 0;
26     struct icm20608_dev *dev = (struct icm20608_dev *
                                       )filp->private_data;
27
28     icm20608_readdata(dev);
29     data[0] = dev->gyro_x_adc;
30     data[1] = dev->gyro_y_adc;
31     data[2] = dev->gyro_z_adc;
32     data[3] = dev->accel_x_adc;
33     data[4] = dev->accel_y_adc;
34     data[5] = dev->accel_z_adc;
```

```
35      data[6] = dev->temp_adc;
36      err = copy_to_user(buf, data, sizeof(data));
37      return 0;
38 }
39
40 /*
41  * @description     : 关闭/释放设备
42  * @param - filp    : 要关闭的设备文件(文件描述符)
43  * @return          : 0 成功;其他 失败
44  */
45 static int icm20608_release(struct inode *inode, struct file *filp)
46 {
47      return 0;
48 }
49
50 /* icm20608 操作函数 */
51 static const struct file_operations icm20608_ops = {
52      .owner = THIS_MODULE,
53      .open = icm20608_open,
54      .read = icm20608_read,
55      .release = icm20608_release,
56 };
```

字符设备驱动框架没什么好说的，重点是第 22~38 行的 icm20608_read 函数，当应用程序调用 read 函数读取 icm20608 设备文件的时候此函数就会执行。此函数调用上面编写好的 icm20608_readdata 函数读取 icm20608 的原始数据并将其上报给应用程序。大家注意，在内核中尽量不要使用浮点运算，所以不要在驱动将 icm20608 的原始值转换为对应的实际值，因为会涉及到浮点计算。

62.5.3 编写测试 APP

新建 icm20608App.c 文件，然后在里面输入如下所示内容：

示例代码 62.5.3.1 icm20608App.c 文件代码

```
1  #include "stdio.h"
2  #include "unistd.h"
3  #include "sys/types.h"
4  #include "sys/stat.h"
5  #include "sys/ioctl.h"
6  #include "fcntl.h"
7  #include "stdlib.h"
8  #include "string.h"
9  #include <poll.h>
10 #include <sys/select.h>
11 #include <sys/time.h>
12 #include <signal.h>
```

```
13 #include <fcntl.h>
14 /***************************************************************
15 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
16 文件名        : icm20608App.c
17 作者          : 左忠凯
18 版本          : V1.0
19 描述          : icm20608 设备测试 APP。
20 其他          : 无
21 使用方法      ：./icm20608App /dev/icm20608
22 论坛          : www.openedv.com
23 日志          : 初版 V1.0 2019/9/20 左忠凯创建
24 ***************************************************************/
25 
26 /*
27  * @description    : main 主程序
28  * @param - argc   : argv 数组元素个数
29  * @param - argv   : 具体参数
30  * @return         : 0 成功;其他 失败
31  */
32 int main(int argc, char *argv[])
33 {
34     int fd;
35     char *filename;
36     signed int databuf[7];
37     unsigned char data[14];
38     signed int gyro_x_adc, gyro_y_adc, gyro_z_adc;
39     signed int accel_x_adc, accel_y_adc, accel_z_adc;
40     signed int temp_adc;
41 
42     float gyro_x_act, gyro_y_act, gyro_z_act;
43     float accel_x_act, accel_y_act, accel_z_act;
44     float temp_act;
45 
46     int ret = 0;
47 
48     if (argc != 2) {
49         printf("Error Usage!\r\n");
50         return -1;
51     }
52 
53     filename = argv[1];
54     fd = open(filename, O_RDWR);
55     if(fd < 0) {
```

```
56          printf("can't open file %s\r\n", filename);
57          return -1;
58      }
59  
60      while (1) {
61          ret = read(fd, databuf, sizeof(databuf));
62          if(ret == 0) {          /* 数据读取成功 */
63              gyro_x_adc = databuf[0];
64              gyro_y_adc = databuf[1];
65              gyro_z_adc = databuf[2];
66              accel_x_adc = databuf[3];
67              accel_y_adc = databuf[4];
68              accel_z_adc = databuf[5];
69              temp_adc = databuf[6];
70  
71              /* 计算实际值 */
72              gyro_x_act = (float)(gyro_x_adc)  / 16.4;
73              gyro_y_act = (float)(gyro_y_adc)  / 16.4;
74              gyro_z_act = (float)(gyro_z_adc)  / 16.4;
75              accel_x_act = (float)(accel_x_adc) / 2048;
76              accel_y_act = (float)(accel_y_adc) / 2048;
77              accel_z_act = (float)(accel_z_adc) / 2048;
78              temp_act = ((float)(temp_adc) - 25 ) / 326.8 + 25;
79  
80              printf("\r\n 原始值:\r\n");
81              printf("gx = %d, gy = %d, gz = %d\r\n", gyro_x_adc,
                        gyro_y_adc, gyro_z_adc);
82              printf("ax = %d, ay = %d, az = %d\r\n", accel_x_adc,
                       accel_y_adc, accel_z_adc);
83              printf("temp = %d\r\n", temp_adc);
84              printf("实际值:");
85              printf("act gx = %.2f°/S, act gy = %.2f°/S,
                        act gz = %.2f°/S\r\n", gyro_x_act, gyro_y_act,
                        gyro_z_act);
86              printf("act ax = %.2fg, act ay = %.2fg,
                       act az = %.2fg\r\n", accel_x_act, accel_y_act,
                       accel_z_act);
87              printf("act temp = %.2f°C\r\n", temp_act);
88          }
89          usleep(100000); /*100ms */
90      }
91      close(fd);  /* 关闭文件 */
92      return 0;
```

```
93 }
```

第 60~91 行，在 while 循环中每隔 100ms 从 icm20608 中读取一次数据，读取到 icm20608 原始数据以后将其转换为实际值，比如陀螺仪就是角速度、加速度计就是 g 值。注意，我们在 icm20608 驱动中将陀螺仪和加速度计的测量范围全部设置到了最大，分别为±2000 和±16g。因此，在计算实际值的时候陀螺仪使用 16.4，加速度计使用 2048。最终将传感器原始数据和得到的实际值显示在终端上。

62.6 运行测试

62.6.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为“icm20608.o”，Makefile 内容如下所示：

示例代码 62.6.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/IMX6ULL/linux/temp/linux-imx-
               rel_imx_4.1.15_2.1.0_ga_alientek
......
4  obj-m := icm20608.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为“icm20608.o”。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“icm20608.ko”的驱动模块文件。

2、编译测试 APP

在 icm20608App.c 这个测试 APP 中我们用到了浮点计算，而 I.MX6U 是支持硬件浮点的，因此我们在编译 icm20608App.c 的时候就可以使能硬件浮点，这样可以加速浮点计算。使能硬件浮点很简单，在编译的时候加入如下参数即可：

```
-march-armv7-a -mfpu-neon -mfloat=hard
```

输入如下命令使能硬件浮点编译 icm20608App.c 这个测试程序：

```
arm-linux-gnueabihf-gcc -march=armv7-a -mfpu=neon -mfloat-abi=hard icm20608App.c -o icm20608App
```

编译成功以后就会生成 icm20608App 这个应用程序，那么究竟有没有使用硬件浮点呢？使用 arm-linux-gnueabihf-readelf 查看一下编译出来的 icm20608App 就知道了，输入如下命令：

```
arm-linux-gnueabihf-readelf -A icm20608App
```

结果如图 62.6.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/22_spi$ arm-linux-gnueabihf-readelf -A icm20608App
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "7-A"
  Tag_CPU_arch: v7
  Tag_CPU_arch_profile: Application
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-2
  Tag_FP_arch: VFPv3
  Tag_Advanced_SIMD_arch: NEONv1
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_rounding: Needed
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align_needed: 8-byte
  Tag_ABI_align_preserved: 8-byte, except leaf SP
  Tag_ABI_enum_size: int
  Tag_ABI_HardFP_use: SP and DP
  Tag_ABI_VFP_args: VFP registers
  Tag_CPU_unaligned_access: v6
zuozhongkai@ubuntu:~/linux/IMX6ULL/Drivers/Linux_Drivers/22_spi$
```

图 62.6.1.1 icm20608App 文件信息

从图 62.6.1.1 可以看出 FPU 架构为 VFPv3，SIMD 使用了 NEON，并且使用了 SP 和 DP，说明 icm20608App 这个应用程序使用了硬件浮点。

62.6.2 运行测试

将上一小节编译出来 icm20608.ko 和 icm20608App 这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，重启开发板，进入到目录 lib/modules/4.1.15 中。输入如下命令加载 icm20608.ko 这个驱动模块。

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe icm20608.ko  //加载驱动模块
```

当驱动模块加载成功以后使用 icm20608App 来测试，输入如下命令：

```
./icm20608App /dev/icm20608
```

测试 APP 会不断的从 ICM20608 中读取数据，然后输出到终端上，如图 62.6.2.1 所示：

```
原始值:
gx = 14, gy = 9, gz = 2
ax = 46, ay = 41, az = 2020
temp = 2909
实际值:act gx = 0.85°/S, act gy = 0.55°/S, act gz = 0.12°/S
act ax = 0.02g, act ay = 0.02g, act az = 0.99g
act temp = 33.82°C
```

图 62.6.2.1 获取到的 ICM20608 数据

可以看出，开发板静止状态下，Z 轴方向的加速度在 1g 左右，这个就是重力加速度。对于陀螺仪来讲，静止状态下三轴的角速度应该在 0° /S 左右。ICM20608 内温度传感器采集到的温度在 30 多度左右，大家可以晃动一下开发板，这个时候陀螺仪和加速度计的值就会有变化。

第六十三章 Linux RS232/485/GPS 驱动实验

串口是很常用的一个外设，在 Linux 下通常通过串口和其他设备或传感器进行通信，根据电平的不同，串口分为 TTL 和 RS232。不管是什么样的接口电平，其驱动程序都是一样的，通过外接 RS485 这样的芯片就可以将串口转换为 RS485 信号，正点原子的 I.MX6U-ALPHA 开发板就是这么做的。对于正点原子的 I.MX6U-ALPHA 开发板而言， RS232、RS485 以及 GPS 模块接口通通连接到了 I.MX6U 的 UART3 接口上，因此这些外设最终都归结为 UART3 的串口驱动。本章我们就来学习一下如何驱动 I.MX6U-ALPHA 开发板上的 UART3 串口，进而实现 RS232、RS485 以及 GSP 驱动。

63.1 Linux 下 UART 驱动框架

1、uart_driver 注册与注销

同 I2C、SPI 一样，Linux 也提供了串口驱动框架，我们只需要按照相应的串口框架编写驱动程序即可。串口驱动没有什么主机端和设备端之分，就只有一个串口驱动，而且这个驱动也已经由 NXP 官方已经编写好了，我们真正要做的就是在设备树中添加所要使用的串口节点信息。当系统启动以后串口驱动和设备匹配成功，相应的串口就会被驱动起来，生成/dev/ttymxcX(X=0….n)文件。

虽然串口驱动不需要我们去写，但是串口驱动框架我们还是需要了解的，uart_driver 结构体表示 UART 驱动，uart_driver 定义在 include/linux/serial_core.h 文件中，内容如下：

示例代码 63.1.1 uart_driver 结构体

```
295 struct uart_driver {
296     struct module     *owner;          /* 模块所属者   */
297     const char        *driver_name;    /* 驱动名字     */
298     const char        *dev_name;       /* 设备名字     */
299     int               major;           /* 主设备号     */
300     int               minor;           /* 次设备号     */
301     int               nr;              /* 设备数       */
302     struct console    *cons;           /* 控制台       */
303 
304     /*
305      * these are private; the low level driver should not
306      * touch these; they should be initialised to NULL
307      */
308     struct uart_state   *state;
309     struct tty_driver   *tty_driver;
310 };
```

每个串口驱动都需要定义一个 uart_driver，加载驱动的时候通过 uart_register_driver 函数向系统注册这个 uart_driver，此函数原型如下：

```
int uart_register_driver(struct uart_driver *drv)
```

函数参数和返回值含义如下：

drv：要注册的 uart_driver。

返回值：0，成功；负值，失败。

注销驱动的时候也需要注销掉前面注册的 uart_driver，需要用到 uart_unregister_driver 函数，函数原型如下：

```
void uart_unregister_driver(struct uart_driver *drv)
```

函数参数和返回值含义如下：

drv：要注销的 uart_driver。

返回值：无。

2、uart_port 的添加与移除

uart_port 表示一个具体的 port，uart_port 定义在 include/linux/serial_core.h 文件，内容如下(有省略)：

示例代码 63.1.2 uart_port 结构体

```
117 struct uart_port {
118     spinlock_t        lock;            /* port lock        */
119     unsigned long     iobase;          /* in/out[bwl]      */
120     unsigned char __iomem   *membase;  /* read/write[bwl]  */
......
235     const struct uart_ops   *ops;
236     unsigned int        custom_divisor;
237     unsigned int        line;          /* port index       */
238     unsigned int        minor;
239     resource_size_t     mapbase;       /* for ioremap      */
240     resource_size_t     mapsize;
241     struct device       *dev;          /* parent device    */
......
250 };
```

uart_port 中最主要的就是第 235 行的 ops，ops 包含了串口的具体驱动函数，这个我们稍后再看。每个 UART 都有一个 uart_port，那么 uart_port 是怎么和 uart_driver 结合起来的呢？这里要用到 uart_add_one_port 函数，函数原型如下：

```
int uart_add_one_port(struct uart_driver      *drv,
                      struct uart_port        *uport)
```

函数参数和返回值含义如下：

drv：此 port 对应的 uart_driver。

uport：要添加到 uart_driver 中的 port。

返回值：0，成功；负值，失败。

卸载 UART 驱动的时候也需要将 uart_port 从相应的 uart_driver 中移除，需要用到 uart_remove_one_port 函数，函数原型如下：

```
int uart_remove_one_port(struct uart_driver *drv, struct uart_port *uport)
```

函数参数和返回值含义如下：

drv：要卸载的 port 所对应的 uart_driver。

uport：要卸载的 uart_port。

返回值：0，成功；负值，失败。

3、uart_ops 实现

在上面讲解 uart_port 的时候说过，uart_port 中的 ops 成员变量很重要，因为 ops 包含了针对 UART 具体的驱动函数，Linux 系统收发数据最终调用的都是 ops 中的函数。ops 是 uart_ops 类型的结构体指针变量，uart_ops 定义在 include/linux/serial_core.h 文件中，内容如下：

示例代码 63.1.3 uart_ops 结构体

```
49 struct uart_ops {
50  unsigned int    (*tx_empty)(struct uart_port *);
51  void        (*set_mctrl)(struct uart_port *, unsigned int mctrl);
52  unsigned int    (*get_mctrl)(struct uart_port *);
53  void        (*stop_tx)(struct uart_port *);
54  void        (*start_tx)(struct uart_port *);
55  void        (*throttle)(struct uart_port *);
```

```
56 void        (*unthrottle)(struct uart_port *);
57 void        (*send_xchar)(struct uart_port *, char ch);
58 void        (*stop_rx)(struct uart_port *);
59 void        (*enable_ms)(struct uart_port *);
60 void        (*break_ctl)(struct uart_port *, int ctl);
61 int         (*startup)(struct uart_port *);
62 void        (*shutdown)(struct uart_port *);
63 void        (*flush_buffer)(struct uart_port *);
64 void        (*set_termios)(struct uart_port *, struct ktermios *new,
65                  struct ktermios *old);
66 void        (*set_ldisc)(struct uart_port *, struct ktermios *);
67 void        (*pm)(struct uart_port *, unsigned int state,
68              unsigned int oldstate);
69 
70 /*
71  * Return a string describing the type of the port
72  */
73 const char  *(*type)(struct uart_port *);
74 
75 /*
76  * Release IO and memory resources used by the port.
77  * This includes iounmap if necessary.
78  */
79 void        (*release_port)(struct uart_port *);
80 
81 /*
82  * Request IO and memory resources used by the port.
83  * This includes iomapping the port if necessary.
84  */
85 int     (*request_port)(struct uart_port *);
86 void        (*config_port)(struct uart_port *, int);
87 int     (*verify_port)(struct uart_port *, struct serial_struct *);
88 int     (*ioctl)(struct uart_port *, unsigned int, unsigned long);
89 #ifdef CONFIG_CONSOLE_POLL
90 int     (*poll_init)(struct uart_port *);
91 void        (*poll_put_char)(struct uart_port *, unsigned char);
92 int     (*poll_get_char)(struct uart_port *);
93 #endif
94 };
```

UART 驱动编写人员需要实现 uart_ops，因为 uart_ops 是最底层的 UART 驱动接口，是实实在在的和 UART 寄存器打交道的。关于 uart_ops 结构体中的这些函数的具体含义请参考 Documentation/serial/driver 这个文档。

UART 驱动框架大概就是这些，接下来我们理论联系实际，看一下 NXP 官方的 UART 驱

动文件是如何编写的。

63.2 I.MX6U UART 驱动分析

1、UART 的 platform 驱动框架

打开 imx6ull.dtsi 文件，找到 UART3 对应的子节点，子节点内容如下所示：

示例代码 63.2.1 uart3 设备节点

```
1  uart3: serial@021ec000 {
2      compatible = "fsl,imx6ul-uart",
3              "fsl,imx6q-uart", "fsl,imx21-uart";
4      reg = <0x021ec000 0x4000>;
5      interrupts = <GIC_SPI 28 IRQ_TYPE_LEVEL_HIGH>;
6      clocks = <&clks IMX6UL_CLK_UART3_IPG>,
7          <&clks IMX6UL_CLK_UART3_SERIAL>;
8      clock-names = "ipg", "per";
9      dmas = <&sdma 29 4 0>, <&sdma 30 4 0>;
10     dma-names = "rx", "tx";
11     status = "disabled";
12 };
```

重点看一下第 2，3 行的 compatible 属性，这里一共有三个值："fsl,imx6ul-uart"、"fsl,imx6q-uar" 和 "fsl,imx21-uart"。在 uboot 源码中搜索这三个值即可找到对应的 UART 驱动文件，此文件为 drivers/tty/serial/imx.c，在此文件中可以找到如下内容：

示例代码 63.2.2 UART platform 驱动框架

```
267 static struct platform_device_id imx_uart_devtype[] = {
268     {
269         .name = "imx1-uart",
270         .driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX1_UART],
271     }, {
272         .name = "imx21-uart",
273         .driver_data = (kernel_ulong_t)
                        &imx_uart_devdata[IMX21_UART],
274     }, {
275         .name = "imx6q-uart",
276         .driver_data = (kernel_ulong_t)
                   &imx_uart_devdata[IMX6Q_UART],
277     }, {
278         /* sentinel */
279     }
280 };
281 MODULE_DEVICE_TABLE(platform, imx_uart_devtype);
282
283 static const struct of_device_id imx_uart_dt_ids[] = {
284     { .compatible = "fsl,imx6q-uart", .data =
```

```
                              &imx_uart_devdata[IMX6Q_UART], },
285     { .compatible = "fsl,imx1-uart", .data =
                              &imx_uart_devdata[IMX1_UART], },
286     { .compatible = "fsl,imx21-uart", .data =
                                &imx_uart_devdata[IMX21_UART], },
287     { /* sentinel */ }
288 };
......
2071 static struct platform_driver serial_imx_driver = {
2072    .probe      = serial_imx_probe,
2073    .remove     = serial_imx_remove,
2074
2075    .suspend    = serial_imx_suspend,
2076    .resume     = serial_imx_resume,
2077    .id_table = imx_uart_devtype,
2078    .driver     = {
2079        .name   = "imx-uart",
2080        .of_match_table = imx_uart_dt_ids,
2081    },
2082 };
2083
2084 static int __init imx_serial_init(void)
2085 {
2086    int ret = uart_register_driver(&imx_reg);
2087
2088    if (ret)
2089        return ret;
2090
2091    ret = platform_driver_register(&serial_imx_driver);
2092    if (ret != 0)
2093        uart_unregister_driver(&imx_reg);
2094
2095    return ret;
2096 }
2097
2098 static void __exit imx_serial_exit(void)
2099 {
2100    platform_driver_unregister(&serial_imx_driver);
2101    uart_unregister_driver(&imx_reg);
2102 }
2103
2104 module_init(imx_serial_init);
2105 module_exit(imx_serial_exit);
```

可以看出 I.MX6U 的 UART 本质上是一个 platform 驱动，第 267~280 行，imx_uart_devtype 为传统匹配表。

第 283~288 行，设备树所使用的匹配表，第 284 行的 compatible 属性值为"fsl,imx6q-uart"。

第 2071~2082 行，platform 驱动框架结构体 serial_imx_driver。

第 2084~2096 行，驱动入口函数，第 2086 行调用 uart_register_driver 函数向 Linux 内核注册 uart_driver，在这里就是 imx_reg。

第 2098~2102 行，驱动出口函数，第 2101 行调用 uart_unregister_driver 函数注销掉前面注册的 uart_driver，也就是 imx_reg。

2、uart_driver 初始化

在 imx_serial_init 函数中向 Linux 内核注册了 imx_reg，imx_reg 就是 uart_driver 类型的结构体变量，imx_reg 定义如下：

示例代码 63.2.3 imx_reg 结构体变量

```
1836 static struct uart_driver imx_reg = {
1837     .owner              = THIS_MODULE,
1838     .driver_name        = DRIVER_NAME,
1839     .dev_name           = DEV_NAME,
1840     .major              = SERIAL_IMX_MAJOR,
1841     .minor              = MINOR_START,
1842     .nr                 = ARRAY_SIZE(imx_ports),
1843     .cons               = IMX_CONSOLE,
1844 };
```

3、uart_port 初始化与添加

当 UART 设备和驱动匹配成功以后 serial_imx_probe 函数就会执行，此函数的重点工作就是初始化 uart_port，然后将其添加到对应的 uart_driver 中。在看 serial_imx_probe 函数之前先来看一下 imx_port 结构体，imx_port 是 NXP 为 I.MX 系列 SOC 定义的一个设备结构体，此结构体内部就包含了 uart_port 成员变量，imx_port 结构体内容如下所示(有缩减)：

示例代码 63.2.4 imx_port 结构体

```
216 struct imx_port {
217     struct uart_port     port;
218     struct timer_list    timer;
219     unsigned int         old_status;
220     unsigned int         have_rtscts:1;
221     unsigned int         dte_mode:1;
222     unsigned int         irda_inv_rx:1;
223     unsigned int         irda_inv_tx:1;
224     unsigned short       trcv_delay; /* transceiver delay */
......
243     unsigned long        flags;
245 };
```

第 217 行，uart_port 成员变量 port。

接下来看一下 serial_imx_probe 函数，函数内容如下：

示例代码 63.2.5 serial_imx_probe 函数

```
1969 static int serial_imx_probe(struct platform_device *pdev)
1970 {
1971     struct imx_port *sport;
1972     void __iomem *base;
1973     int ret = 0;
1974     struct resource *res;
1975     int txirq, rxirq, rtsirq;
1976 
1977     sport = devm_kzalloc(&pdev->dev, sizeof(*sport), GFP_KERNEL);
1978     if (!sport)
1979         return -ENOMEM;
1980 
1981     ret = serial_imx_probe_dt(sport, pdev);
1982     if (ret > 0)
1983         serial_imx_probe_pdata(sport, pdev);
1984     else if (ret < 0)
1985         return ret;
1986 
1987     res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
1988     base = devm_ioremap_resource(&pdev->dev, res);
1989     if (IS_ERR(base))
1990         return PTR_ERR(base);
1991 
1992     rxirq = platform_get_irq(pdev, 0);
1993     txirq = platform_get_irq(pdev, 1);
1994     rtsirq = platform_get_irq(pdev, 2);
1995 
1996     sport->port.dev = &pdev->dev;
1997     sport->port.mapbase = res->start;
1998     sport->port.membase = base;
1999     sport->port.type = PORT_IMX,
2000     sport->port.iotype = UPIO_MEM;
2001     sport->port.irq = rxirq;
2002     sport->port.fifosize = 32;
2003     sport->port.ops = &imx_pops;
2004     sport->port.rs485_config = imx_rs485_config;
2005     sport->port.rs485.flags =
2006         SER_RS485_RTS_ON_SEND | SER_RS485_RX_DURING_TX;
2007     sport->port.flags = UPF_BOOT_AUTOCONF;
2008     init_timer(&sport->timer);
2009     sport->timer.function = imx_timeout;
2010     sport->timer.data     = (unsigned long)sport;
2011 
```

```
2012    sport->clk_ipg = devm_clk_get(&pdev->dev, "ipg");
2013    if (IS_ERR(sport->clk_ipg)) {
2014        ret = PTR_ERR(sport->clk_ipg);
2015        dev_err(&pdev->dev, "failed to get ipg clk: %d\n", ret);
2016        return ret;
2017    }
2018
2019    sport->clk_per = devm_clk_get(&pdev->dev, "per");
2020    if (IS_ERR(sport->clk_per)) {
2021        ret = PTR_ERR(sport->clk_per);
2022        dev_err(&pdev->dev, "failed to get per clk: %d\n", ret);
2023        return ret;
2024    }
2025
2026    sport->port.uartclk = clk_get_rate(sport->clk_per);
2027    if (sport->port.uartclk > IMX_MODULE_MAX_CLK_RATE) {
2028        ret = clk_set_rate(sport->clk_per, IMX_MODULE_MAX_CLK_RATE);
2029        if (ret < 0) {
2030            dev_err(&pdev->dev, "clk_set_rate() failed\n");
2031            return ret;
2032        }
2033    }
2034    sport->port.uartclk = clk_get_rate(sport->clk_per);
2035
2036    /*
2037     * Allocate the IRQ(s) i.MX1 has three interrupts whereas later
2038     * chips only have one interrupt.
2039     */
2040    if (txirq > 0) {
2041        ret = devm_request_irq(&pdev->dev, rxirq, imx_rxint, 0,
2042                       dev_name(&pdev->dev), sport);
2043        if (ret)
2044            return ret;
2045
2046        ret = devm_request_irq(&pdev->dev, txirq, imx_txint, 0,
2047                       dev_name(&pdev->dev), sport);
2048        if (ret)
2049            return ret;
2050    } else {
2051        ret = devm_request_irq(&pdev->dev, rxirq, imx_int, 0,
2052                       dev_name(&pdev->dev), sport);
2053        if (ret)
2054            return ret;
```

```
2055    }
2056
2057    imx_ports[sport->port.line] = sport;
2058
2059    platform_set_drvdata(pdev, sport);
2060
2061    return uart_add_one_port(&imx_reg, &sport->port);
2062 }
```

第 1971 行，定义一个 imx_port 类型的结构体指针变量 sport。

第 1977 行，为 sport 申请内存。

第 1987~1988 行，从设备树中获取 I.MX 系列 SOC UART 外设寄存器首地址，对于 I.MX6ULL 的 UART3 来说就是 0X021EC000。得到寄存器首地址以后对其进行内存映射，得到对应的虚拟地址。

第 1992~1994 行，获取中断信息。

第 1996~2034 行，初始化 sport，我们重点关注的就是第 2003 行初始化 sport 的 port 成员变量，也就是设置 uart_ops 为 imx_pops，imx_pops 就是 I.MX6ULL 最底层的驱动函数集合，稍后再来看。

第 2040~2055 行，申请中断。

第 2061 行，使用 uart_add_one_port 向 uart_driver 添加 uart_port，在这里就是向 imx_reg 添加 sport->port。

4、imx_pops 结构体变量

imx_pops 就是 uart_ops 类型的结构体变量，保存了 I.MX6ULL 串口最底层的操作函数，imx_pops 定义如下：

示例代码 63.2.6 imx_pops 结构体

```
1611 static struct uart_ops imx_pops = {
1612    .tx_empty        = imx_tx_empty,
1613    .set_mctrl       = imx_set_mctrl,
1614    .get_mctrl       = imx_get_mctrl,
1615    .stop_tx         = imx_stop_tx,
1616    .start_tx        = imx_start_tx,
1617    .stop_rx         = imx_stop_rx,
1618    .enable_ms       = imx_enable_ms,
1619    .break_ctl       = imx_break_ctl,
1620    .startup         = imx_startup,
1621    .shutdown        = imx_shutdown,
1622    .flush_buffer    = imx_flush_buffer,
1623    .set_termios     = imx_set_termios,
1624    .type            = imx_type,
1625    .config_port     = imx_config_port,
1626    .verify_port     = imx_verify_port,
1627 #if defined(CONFIG_CONSOLE_POLL)
1628    .poll_init       = imx_poll_init,
1629    .poll_get_char   = imx_poll_get_char,
```

```
1630     .poll_put_char    = imx_poll_put_char,
1631 #endif
1632 };
```

imx_pops 中的函数基本都是和 I.MX6ULL 的 UART 寄存器打交道的，这里就不去详细的分析了。简单的了解了 I.MX6U 的 UART 驱动以后我们再来学习一下，如何驱动正点原子 I.MX6U-ALPHA 开发板上的 UART3 接口。

63.3 硬件原理图分析

本实验要用到的 I.MX6U 的 UART3 接口，I.MX6U-ALPHA 开发板上 RS232、RS485 和 GPS 这三个接口都连接到了 UART3 上，我们依次来看一下这三个模块的原理图。

1、RS232 原理图

RS232 原理图如图 63.3.1 所示：

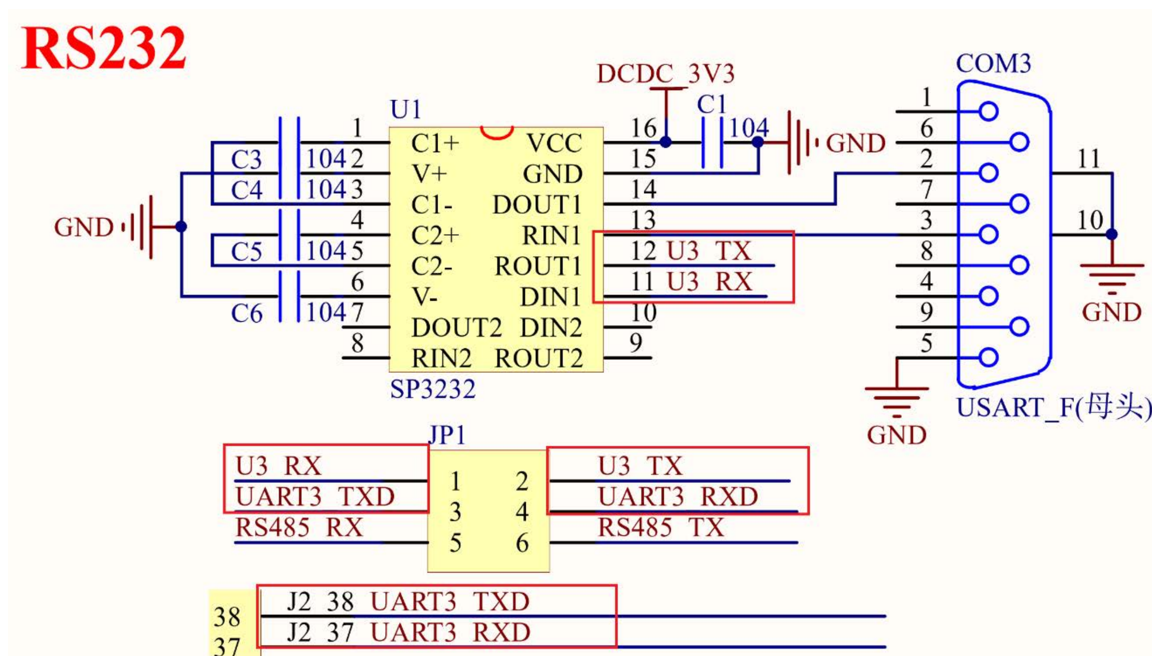


图 63.3.1 RS232 原理图

从图 63.3.1 可以看出，RS232 电平通过 SP3232 这个芯片来实现，RS232 连接到了 I.MX6U 的 UART3 接口上，但是要通过 JP1 这个跳线帽设置。把 JP1 的 1-3 和 2-4 连接起来以后 SP3232 就和 UART3 连接到了一起。

2、RS485 原理图

RS485 原理图如图 63.3.2 所示：

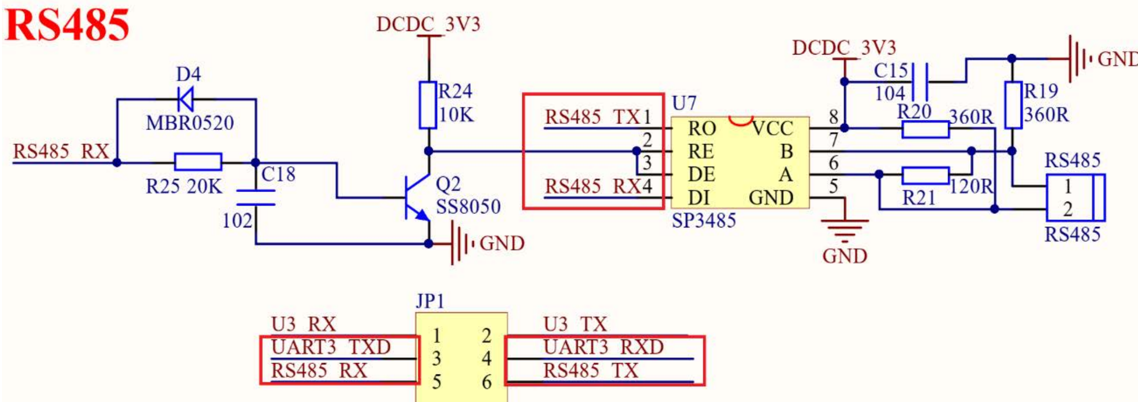


图 63.3.2 RS485 原理图

RS485 采用 SP3485 这颗芯片来实现，RO 为数据输出端，RI 为数据输入端，RE 是接收使能信号(低电平有效)，DE 是发送使能信号(高电平有效)。在图 63.3.2 中 RE 和 DE 经过一系列的电路，最终通过 RS485_RX 来控制，这样我们可以省掉一个 RS485 收发控制 IO，将 RS485 完全当作一个串口来使用，方便我们写驱动。

3、GPS 原理图

正点原子有一款 GPS+北斗定位模块，型号为 ATK1218-BD，I.MX6U-ALPHA 开发板留出了这款 GPS 定位模块的接口，接口原理图如图 63.3.3 所示：

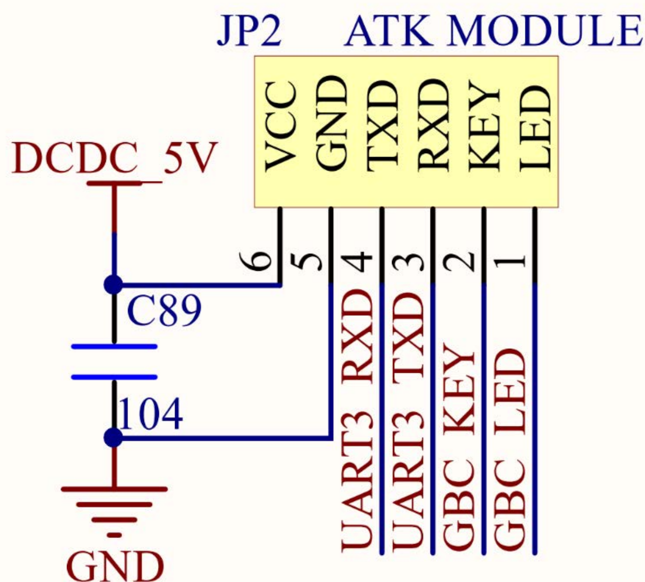


图 63.3.3 ATK MODULE 模块。

从图 63.3.3 可以看出，GPS 模块用的也是 UART3，因此 UART3 驱动成功以后就可以直接读取 GPS 模块数据了。

63.4 RS232 驱动编写

前面我们已经说过了，I.MX6U 的 UART 驱动 NXP 已经编写好了，所以不需要我们编写。

我们要做的就是在设备树中添加 UART3 对应的设备节点即可。打开 imx6ull-alientek-emmc.dts 文件，在此文件中只有 UART1 对应的 uart1 节点，并没有 UART3 对应的节点，因此我们可以参考 uart1 节点创建 uart3 节点。

1、UART3 IO 节点创建

UART3 用到了 UART3_TXD 和 UART3_RXD 这两个 IO，因此要先在 iomuxc 中创建 UART3 对应的 pinctrl 子节点，在 iomuxc 中添加如下内容：

示例代码 63.4.1 UART3 引脚 pinctrl 节点

```
1 pinctrl_uart3: uart3grp {
2     fsl,pins = <
3         MX6UL_PAD_UART3_TX_DATA__UART3_DCE_TX      0X1b0b1
4         MX6UL_PAD_UART3_RX_DATA__UART3_DCE_RX      0X1b0b1
5     >;
6 };
```

最后检查一下 UART3_TX 和 UART3_RX 这两个引脚有没有被用作其他功能，如果有的话要将其屏蔽掉，保证这两个 IO 只用作 UART3，切记!!!

2、添加 uart3 节点

默认情况下 imx6ull-alientek-emmc.dts 中只有 uart1 和 uart2 这两个节点，如图 63.4.1 所示：

```
789 &uart1 {
790     pinctrl-names = "default";
791     pinctrl-0 = <&pinctrl_uart1>;
792     status = "okay";
793 };
794
795 &uart2 {
796     pinctrl-names = "default";
797     pinctrl-0 = <&pinctrl_uart2>;
798     fsl,uart-has-rtscts;
799     /* for DTE mode, add below change */
800     /* fsl,dte-mode; */
801     /* pinctrl-0 = <&pinctrl_uart2dte>; */
802     status = "okay";
803 };
```

图 63.4.1 uart1 和 uart2 节点

uart1 是 UART1 的，在正点原子的 I.MX6U-ALPHA 开发板上没有用到 UART2，而且 UART2 默认用到了 UART3 的 IO，因此需要将 uart2 这个节点删除掉，然后加上 UART3 对应的 uart3，uart3 节点内容如下：

示例代码 63.4.2 UART3 对应的 uart3 节点

```
1 &uart3 {
2     pinctrl-names = "default";
3     pinctrl-0 = <&pinctrl_uart3>;
4     status = "okay";
5 };
```

完成以后重新编译设备树并使用新的设备树启动 Linux，如果设备树修改成功的话，系统启动以后就会生成一个名为“/dev/ttymxc2”的设备文件，ttymxc2 就是 UART3 对应的设备文件，应用程序可以通过访问 ttymxc2 来实现对 UART3 的操作。

63.5 移植 minicom

minicom 类似我们常用的串口调试助手，是 Linux 下很常用的一个串口工具，将 minicom 移植到我们的开发板中，这样我们就可以借助 minicom 对串口进行读写操作。

1、移植 ncurses

minicom 需要用到 ncurses，依次需要先移植 ncurses，如果前面已经移植好了 ncurses，那么这里就不需要再次移植了，只需要在编译 minicom 的时候指定 ncurses 库和头文件目录 即可。

首先在 ubuntu 中创建一个目录来存放我们要移植的文件，比如我在 /home/zuozhongkai/linux/IMX6ULL 目录下创建了一个名为“tool”的目录来存放所有的移植文件。然后下载 ncurses 源码，我们已经将 ncurses 源码放到了开发板光盘中，路径为：1、例程源码-》7、第三方库源码-》ncurses-6.0.tar.gz，将 ncurses-6.0.tar.gz 拷贝到 Ubuntu 中创建的 tool 目录下，然后进行解压，解压命令如下：

```
tar -vxzf ncurses-6.0.tar.gz
```

解压完成以后就会生成一个名为“ncurses-6.0”的文件夹，此文件夹就是 ncureses 的源码文件夹。在 tool 目录下新建名为“ncurses”目录，用于保存 ncurses 编译结果，一切准备就绪以后就可以编译 ncureses 库了。进入到 ncureses 源码目录下，也就是刚刚解压出来的 ncurses-6.0 目录中，首先是配置 ncureses，输入如下命令：

```
./configure --prefix=/home/zuozhongkai/linux/IMX6ULL/tool/ncurses --host=arm-linux-gnueabihf --with-shared
```

configure 就是配置脚本，--prefix 用于指定编译结果的保存目录，这里肯定将编译结果保存到我们前面创建的“ncurses”目录中。--hsot 用于指定编译器前缀，这里设置为 “arm-linux-gnueabihf”。配置命令写好以后点击回车键，等待配置完成，配置成功以后如图 63.5.1 所示：

```
** Configuration summary for NCURSES 6.0 20150808:

       extended funcs: yes
       xterm terminfo: xterm-new

        bin directory: /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/bin
        lib directory: /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/lib
    include directory: /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
        man directory: /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/share/man
   terminfo directory: /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/share/terminfo

** Include-directory is not in a standard location
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/ncurses-6.0$ ls
```

图 63.5.1 配置成功

配置成功以后输入“make”命令开始编译，编译成功以后如图 63.5.2 所示：

```
make[1]: Leaving directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/test'
cd misc && make DESTDIR="" RPATH_LIST="/home/zuozhongkai/linux/IMX6ULL/tool/ncurses/lib" all
make[1]: Entering directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/misc'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/misc'
cd c++ && make DESTDIR="" RPATH_LIST="/home/zuozhongkai/linux/IMX6ULL/tool/ncurses/lib" all
make[1]: Entering directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/c++'
arm-linux-gnueabihf-g++  -o demo ../obj_s/demo.o -L../lib -lncurses++ -L../lib -lform -lmenu -lpanel
-lncurses    -lutil  -Wl,-rpath,/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/lib -lstdc++ -DHAVE_
CONFIG_H -I. -I../include  -D_GNU_SOURCE -D_FILE_OFFSET_BITS=64  -DNDEBUG -O2  -fPIC
make[1]: Leaving directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/c++'
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/ncurses-6.0$
```

图 63.5.2 编译成功

编译成功以后输入“make install”命令安装，安装的意思就是将编译出来的结果拷贝到--pfefix 指定的目录里面去。安装成功以后如图 63.5.3 所示：

```
arm-linux-gnueabihf-ranlib /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/lib/libncurses++_g.a
installing ./cursesapp.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing ./cursesf.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing ./cursesm.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing ./cursesp.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing ./cursesw.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing ./cursslk.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
installing etip.h in /home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include/ncurses
make[1]: Leaving directory '/home/zuozhongkai/linux/IMX6ULL/tool/ncurses-6.0/c++'
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/ncurses-6.0$
```

图 63.5.3 安装成功

安装成功以后查看一下前面创建的“ncurses”文件夹，会发现里面多了一些东西，如图 63.5.4 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/ncurses$ ls
bin  include  lib  share
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/ncurses$
```

图 63.5.4 编译出来的结果

我们需要将图 63.5.4 中 include、lib 和 share 这三个目录中存放的文件分别拷贝到开发板根文件系统中的/usr/include、/usr/lib 和/usr/share 这三个目录中，如果哪个目录不存在的话请自行创建！！拷贝命令如下：

```
sudo cp lib/* /home/zuozhongkai/linux/nfs/rootfs/usr/lib/ -rfa
sudo cp share/* /home/zuozhongkai/linux/nfs/rootfs/usr/share/ -rfa
sudo cp include/* /home/zuozhongkai/linux/nfs/rootfs/usr/include/ -rfa
```

然后在开发板根目录的/etc/profile(没有的话自己创建一个)文件中添加如下所示内容：

示例代码 63.5.1 /etc/provile 文件

```
1 #!/bin/sh
2 LD_LIBRARY_PATH=/lib:/usr/lib:$LD_LIBRARY_PATH
3 export LD_LIBRARY_PATH
4
5 export TERM=vt100
6 export TERMINFO=/usr/share/terminfo
```

2、移植 minicom

继续移植 minicom，获取 minicom 源码，我们已经放到了开发板光盘中了，路径为：1、例程源码-》7、第三方库源码-》minicom-2.7.1.tar.gz。将 minicom-2.7.1.tar.gz 拷贝到 ubuntu 中的/home/zuozhongkai/linux/IMX6ULL/tool 目录下，然后在 tool 目录下新建一个名为“minicom”的子目录，用于存放 minicom 编译结果。一切准备好以后就可以编译 minicom 了，先解压 minicom，命令如下：

```
tar -vxzf minicom-2.7.1.tar.gz
```

解压完成以后会生成一个叫做 minicom-2.7.1 的文件夹，这个就是 minicom 的源码，进入到此目录中，然后配置 minicom，配置命令如下：

```
cd minicom-2.7.1/                //进入 minicom 源码目录
./configure CC=arm-linux-gnueabihf-gcc --prefix=/home/zuozhongkai/linux/IMX6ULL/tool/minicom --host=arm-linux-gnueabihf CPPFLAGS=-I/home/zuozhongkai/linux/IMX6ULL/tool/ncurses/include    LDFLAGS=-L/home/zuozhongkai/linux/IMX6ULL/tool/ncurses/lib  -enable-cfg-dir=/etc/minicom            //配置
```

CC 表示要使用的 gcc 交叉编译器，--prefix 指定编译出来的文件存放目录，肯定要存放到我们前面创建的 minicom 目录中。--host 指定交叉编译器前缀，CPPFLAGS 指定 ncurses 的头文件路径，LDFLAGS 指定 ncurses 的库路径。

配置成功的话如图 63.5.5 所示：

```
config.status: creating lib/Makefile
config.status: creating src/Makefile
config.status: creating po/Makefile.in
config.status: creating minicom.spec
config.status: creating config.h
config.status: executing depfiles commands
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/minicom-2.7.1$
```

图 63.5.5 配置成功

配置成功以后执行如下命令编译并安装：

```
make
make install
```

编译安装完成以后，前面创建的 minicom 目录内容如图 63.5.6 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/minicom$ ls
bin  share
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/minicom$
```

图 63.5.6 minicom 安装编译结果

将 minicom 目录中 bin 子目录下的所有文件拷贝到开发板根目录中的/usr/bin 目录下，命令如下：

```
sudo cp bin/* /home/zuozhongkai/linux/nfs/rootfs/usr/bin/
```

完成以后在开发板中输入"minicom -v"来查看 minicom 工作是否正常，结果如图 63.5.7 所示：

```
/etc # minicom -v
minicom version 2.7.1 (compiled Sep 13 2019)
Copyright (C) Miquel van Smoorenburg.

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version
2 of the License, or (at your option) any later version.

/etc #
```

图 63.5.7 minicom 版本号

从图 63.5.7 可以看出，此时 minicom 版本号为 2.7.1，minicom 版本号查看正常。输入如下命令打开 minicom 配置界面：

```
minicom -s
```

结果是打不开 minicom 配置界面，提示如图 63.5.8 所示信息：

```
/ # minicom -s
You don't exist. Go away.
/ #
```

图 63.5.8 minicom 打开失败

从图 63.5.8 可以看出，minicom 异常嚣张，竟然让我们"Go away"，这能容忍？！必须要治一下。解决方法很简单，新建/etc/passwd 文件，然后在 passwd 文件里面输入如下所示内容：

示例代码 63.5.2 /etc/passwd 文件

```
1 root:x:0:0:root:/root:/bin/sh
```

完成以后重启开发板！
完成以后重启开发板！
完成以后重启开发板！

开发板重启以后再执行“minicom -s”命令，此时 minicom 配置界面就可以打开了，如图 63.5.9 所示：

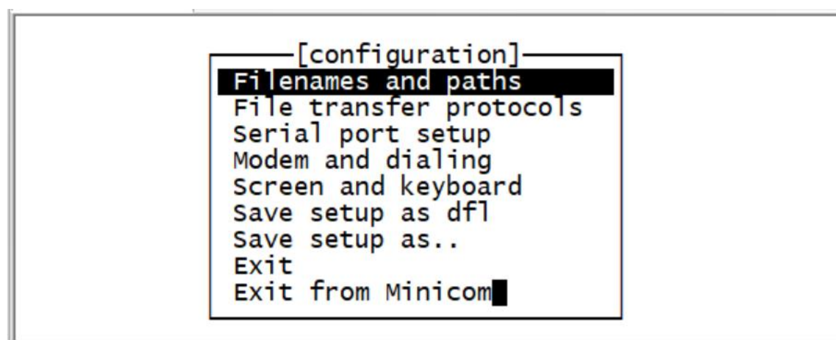


图 63.5.9 mincom 配置界面

如果能出现图 63.5.9 所示界面，那么就说明 mincom 工作正常了。

63.6 RS232 驱动测试

63.6.1 RS232 连接设置

在测试之前要先将 I.MX6U-ALPHA 开发板的 RS232 与电脑连接起来，首先设置 JP1 跳线帽，如图 63.6.1.1 所示：

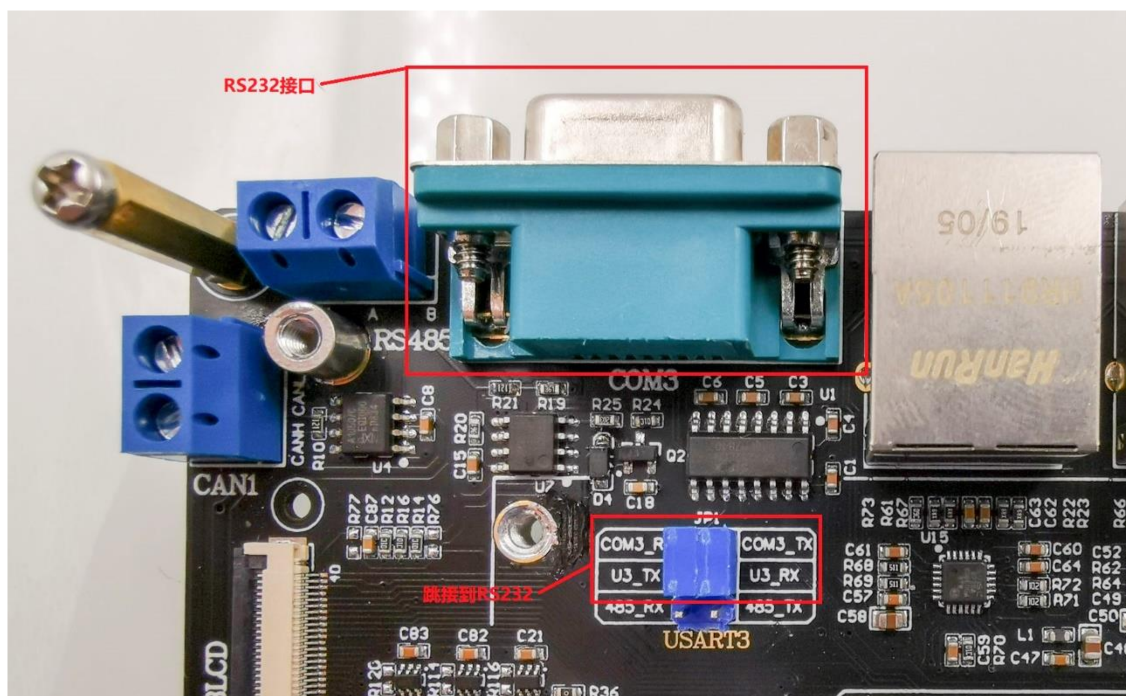


图 63.6.1.1 UART3跳线帽设置

跳线帽设置好以后使用 RS232 线将开发板与电脑连接起来，这里建议使用 USB 转 DB9(RS232)数据线，比如正点原子售卖的 CH340 方案的 USB 转公头 DB9 数据线，如图 63.6.1.2 所示：



图 63.6.1.2 USB 转 DB9 数据线

图 63.6.1.2 中所示的数据线是带有 CH340 芯片的，因此当连接到电脑以后就会出现一个 COM 口，这个 COM 口就是我们要使用的 COM 口。比如在我的电脑上就是 COM9，在 SecureCRT 上新建一个连接，串口为 COM9，波特率为 115200。

63.6.2 minicom 设置

在开发板中输入“minicom -s”，打开 minicom 配置界面，然后选中“Serial port setup”，如图 63.6.2.1 所示：

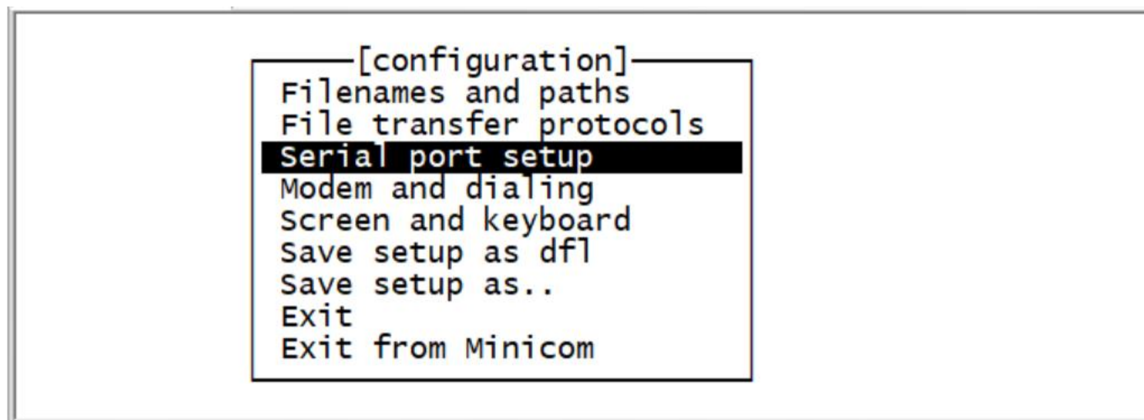


图 63.6.2.1 选中串口设置项

选中“Serial port setup”以后点击回车，进入设置菜单，如图 63.6.2.2 所示：

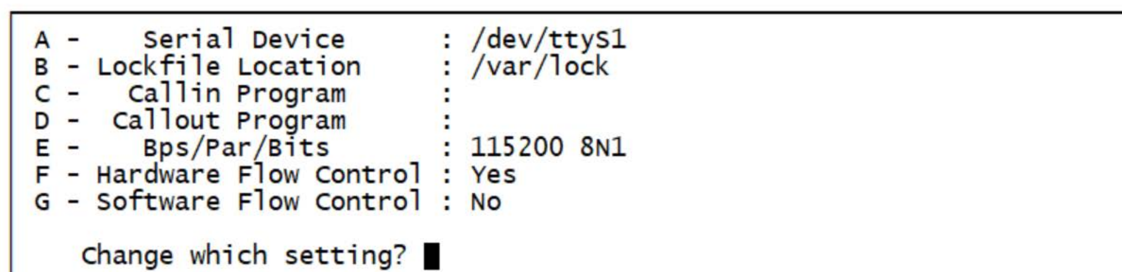


图 63.6.2.2 串口设置项

图 63.6.2.2 中有 7 个设置项目，分别对应 A、B……G，比如第一个是选中串口，UART3 的串口文件为/dev/ttymxc2，因此串口设置要设置为/dev/ttymxc2。设置方法就是按下键盘上的‘A’，然后输入“/dev/ttymxc2”即可，如图 63.6.2.3 所示：

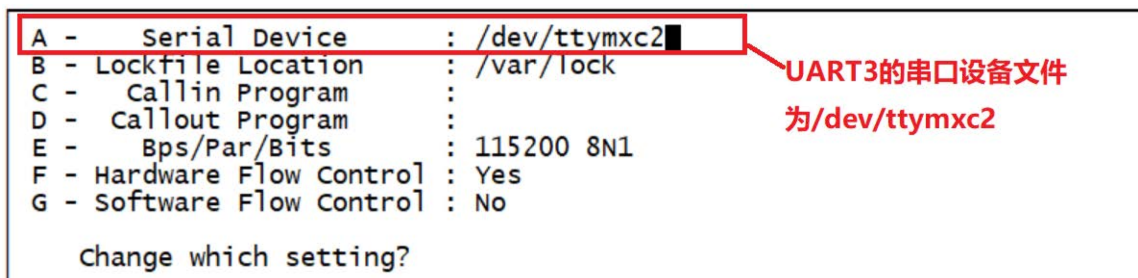


图 63.6.2.3 串口设备文件设置

设置完以后按下回车键确认，确认完以后就可以设置其他的配置项。比如 E 设置波特率、数据位和停止位的、F 设置硬件流控的，设置方法都一样，设置完以后如图 63.6.2.4 所示：

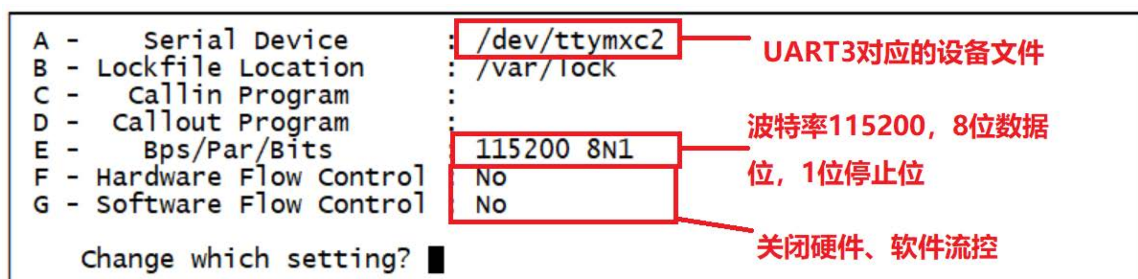


图 63.6.2.4 UART3 设置

都设置完成以后按下回车键确认并退出，这时候会退回到如图 63.6.2.1 所示的界面，按下 ESC 键退出图 63.6.2.1 所示的配置界面，退出以后如图 63.6.2.5 所示：

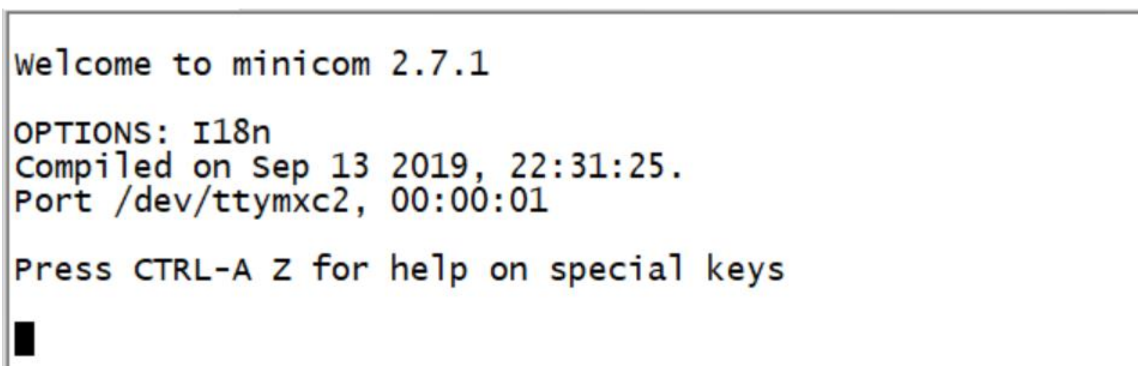


图 63.6.2.5 minicom 串口界面

图 63.6.2.2 就是我们的串口调试界面，可以看出当前的串口文件为/dev/ttymxc2，按下 CTRL-A，然后再按下 Z 就可以打开 minicom 帮助信息界面，如图 63.6.2.6 所示：

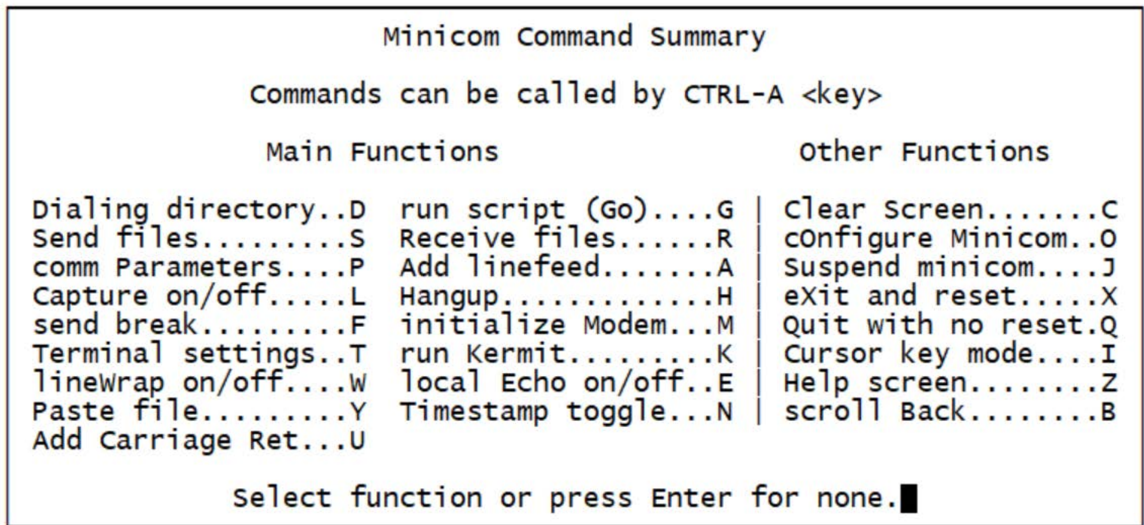


图 63.6.2.6 minicom 帮助信息界面

从图 63.6.2.6 可以看出，minicom 有很多快捷键，本实验我们打开 minicom 的回显功能，回显功能配置项为“local Echo on/off..E”，因此按下 E 即可打开/关闭回显功能。

63.6.3 RS232 收发测试

1、发送测试

首先测试开发板通过 UART3 向电脑发送数据的功能，需要打开 minicom 的回显功能(不打开也可以，但是在 minicom 中看不到自己输入的内容)，回显功能打开以后输入“AAAA”，如图 63.6.3.1 所示：

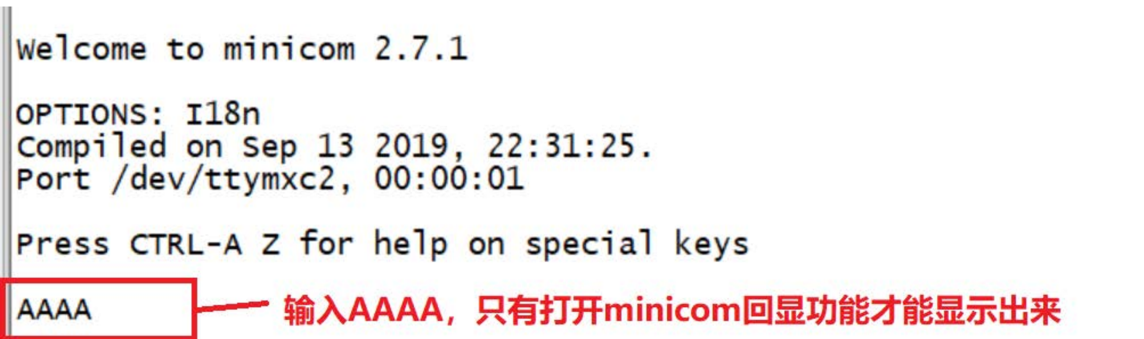


图 63.6.3.1 通过 UART3 向电脑发送“AAAA”

图 63.6.3.1 中的“AAAA”相当于开发板通过 UART3 向电脑发送“AAAA”，那么 COM9 就会接收到“AAAA”，SecureCRT 中 COM9 收到的数据如图 63.6.3.2 所示：



图 63.6.3.2 电脑接收到开发板发送过来的数据

可以看出，开发板通过 UART3 向电脑发送数据正常，那么接下来就测试开发板数据接收功能。

2、接收测试

接下来测试开发板的 UART3 接收功能，同样的，要先打开 SecureCRT 上 COM9 的本地回显，否则的话你在 COM9 上输出的内容会看不到，但是实际上是已经发送给了开发板。选中 SecureCRT 的 Options->Session Options->Adavnced，打开回话配置界面，然后选中“Local echo”，如图 63.6.3.3 所示：

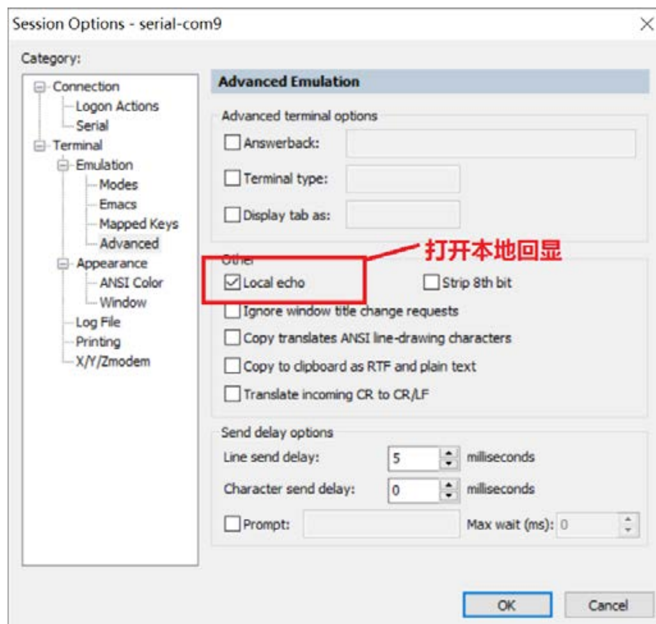


图 63.6.3.3 打开 SecureCRT 的本地回显

SecureCRT 设置好以后向开发板发送一个“BBBB”，在 SecureCRT 的 COM9 上输入“BBBB”，如图 63.6.3.3 所示：



图 63.6.3.3 电脑向开发板发送“BBBB”

此时开发板的 minicom 就会接收到发送过来的“BBBB”，如图 63.6.3.4 所示：



图 63.6.3.4 开发板接收到发送过来的数据

UART3 收发测试都没有问题，说明我们的 UART3 驱动工作正常。如果要退出 minicom 的话，在 minicom 通信界面按下 CRTL+A，然后按下 X 来关闭 minicom。关于 minicom 的使用我们这里讲的很简单，大家可以在网上查找更加详细的 minicom 使用教程。

63.7 RS485 测试

前面已经说过了，I.MX6U-ALPHA 开发板上的 RS485 接口连接到了 UART3 上，因此本质上就是个串口。RS232 实验我们已经将 UART3 的驱动编写好了，所以 RS485 实验就不需要编写任何驱动程序，可以直接使用 minicom 来进行测试。

63.7.1 RS485 连接设置

首先是设置 JP1 跳线帽，将 3-5、4-6 连接起来，如图 63.7.1.1 所示：

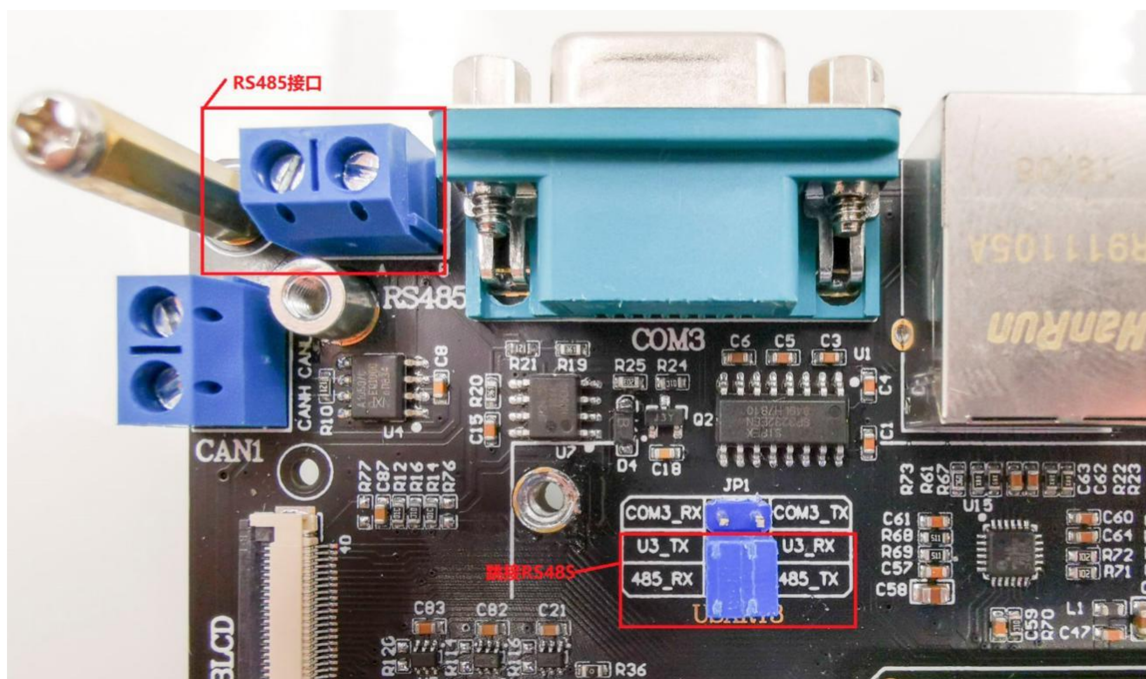


图 63.7.1.1 RS485 接口设置

一个板子是不能进行 RS485 通信测试的，还需要另一个 RS485 设备，比如另外一块 I.MX6U-ALPHA 开发板。这里推荐大家使用正点原子出品的 USB 三合一串口转换器，支持 USB 转 TTL、RS232 和 RS485，如图 63.7.1.2 所示：



图 63.7.1.2 正点原子 USB 三合一串口转换器

使用杜邦线将 USB 串口转换器的 RS485 接口和 I.MX6U-ALPHA 开发板的 RS485 连接起来，A 接 A，B 接 B，不能接错了！连接完成以后如图 63.7.1.3 所示：

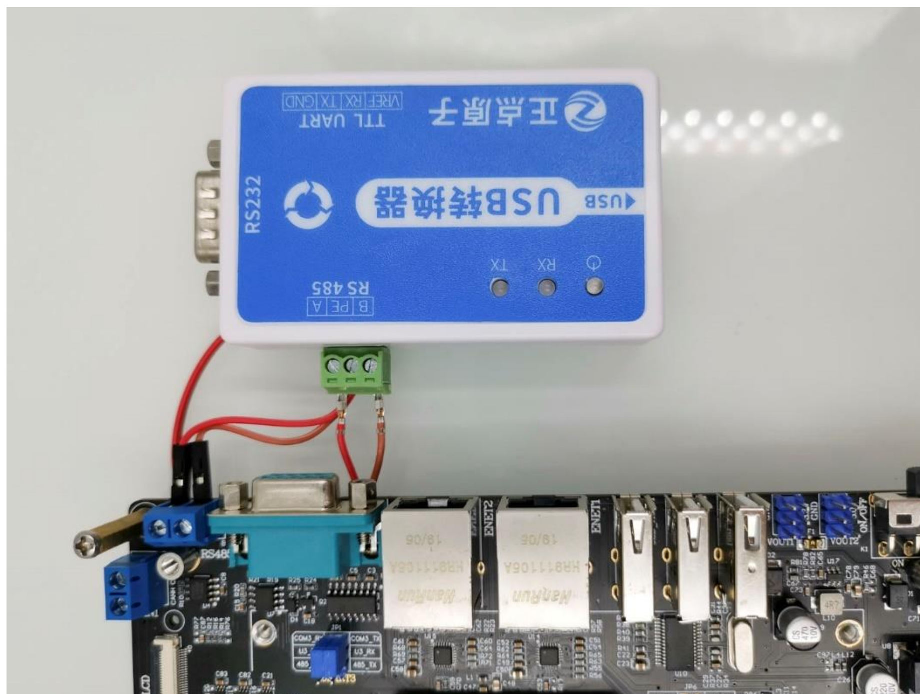


图 63.7.1.3 串口转换器和开发板 RS485 连接示意图

串口转换器通过 USB 线连接到电脑上，我用的是 CH340 版本的，因此就不需要安装驱动的，如果使用的是 FT232 版本的就需要安装相应的驱动。连接成功以后电脑就会有相应的 COM 口，比如我的电脑上就是 COM10，接下来就是测试。

63.7.2 RS485 收发测试

RS485 的测试和 RS232 一模一样!USB 多合一转换器的 COM 口为 10，因此使用 SecureCRT 创建一个 COM10 的连接。开发板使用 UART3，对应的串口设备文件为/dev/ttymxc2，因此开发板使用 minicom 创建一个/dev/ttymxc2 的串口连接。串口波特率都选择 115200，8 位数据位，1 位停止位，关闭硬件和软件流控。

1、RS485 发送测试

首先测试开发板通过 RS485 发送数据，设置好 minicom 以后，同样输入“AAAA”，也就是通过 RS485 向电脑发送一串“AAAA”。如果 RS485 驱动工作正常的话，那么电脑就会介绍到开发板发送过来的“AAAA”，如图 63.7.2.1 所示：

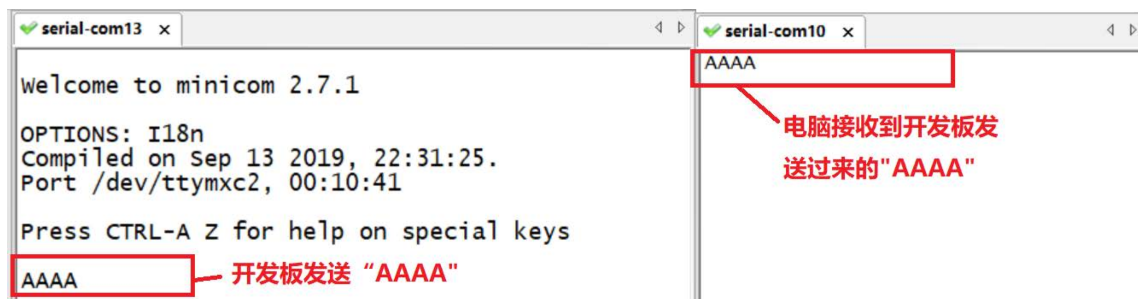


图 63.7.2.1 RS485 数据发送测试

从图 63.7.2.1 可以看出开发板通过 RS485 向电脑发送“AAAA”成功，说明 RS485 数据数据发送正常。

2、RS485 接收测试

接下来测试一下 RS485 数据接收，电脑通过 RS485 向开发板发送“BBBB”，然后观察 minicom 是否能接收到“BBBB”。结果如图 63.7.2.2 所示：

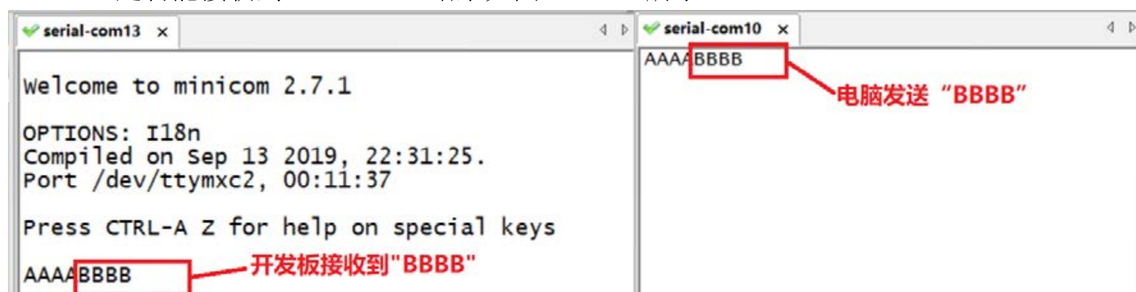


图 63.7.2.2 RS485 数据接收测试

从图 63.7.2.1 可以看出开发板接收到电脑通过 RS485 发送过来的“BBBB”，说明 RS485 数据接收也正常。

63.8 GPS 测试

63.8.1 GPS 连接设置

GPS 模块大部分都是串口输出的，这里以正点原子出品的 ATK1218-BD 模块为例，这是一款 GSP+北斗的定位模块，模块如图 63.8.1.1 所示：



图 63.8.1.1 正点原子 ATK1218-BD 定位模块

首先要将 I.MX6U-ALPHA 开发板上的 JP1 跳线帽拔掉，不能连接 RS232 或 RS485，否则会干扰到 GSP 模块。UART3_TX 和 UART3_RX 已经连接到了开发板上的 ATK MODULE 上，直接将 ATK1218-BD 模块插到开发板上的 ATK MODULE 接口即可，开发板上的 ATK MODULE 接口是 6 脚的，而 ATK1218-BD 模块是 5 脚的，因此需要靠左插！然后 GPS 需要接上天线，天线的接收头一定要放到户外，因此室内一般是没有 GPS 信号的。连接完成以后如图 63.8.1.2 所示：

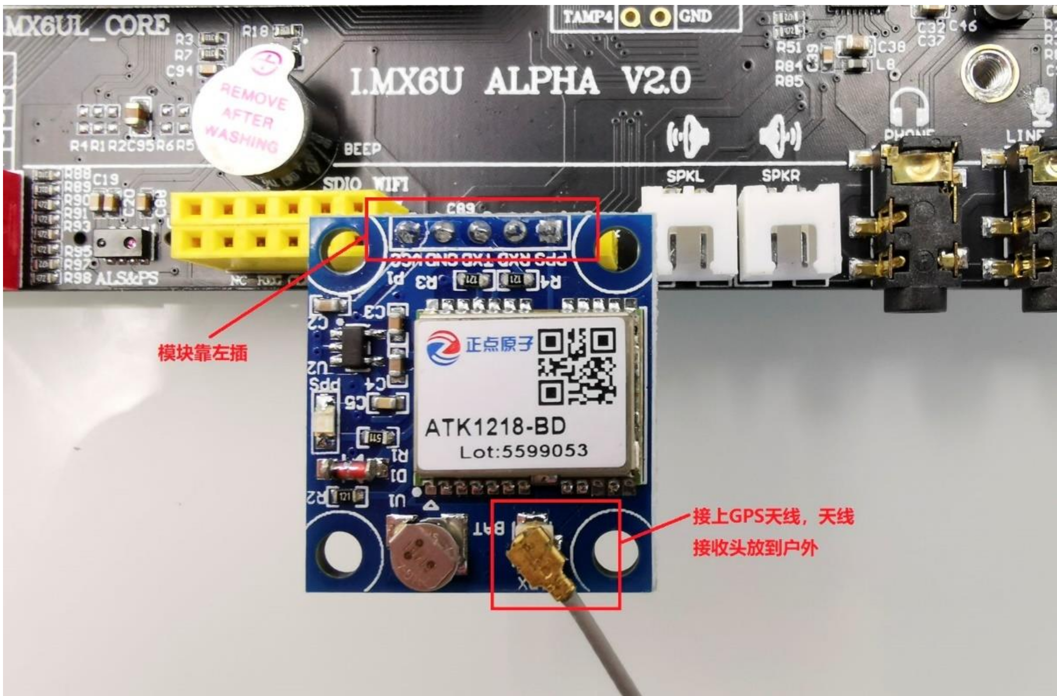


图 63.8.1.2 GPS 模块连接示意图

63.8.2 GPS 数据接收测试

GPS 我们都是被动接收定位数据的，因此打开 minicom，设置/dev/ttymxc2，串口设置要求如下：

①、波特率设置为 38400，因为正点原子的 ATK1218-BD 模块默认波特率就是 38400。

②、8 位数据位，1 位停止位。

③、关闭硬件和软件流控。

设置好以后如图 63.8.2.1 所示：

```
A -    Serial Device      : /dev/ttymxc2
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 38400 8N1
F - Hardware Flow Control : Yes
G - Software Flow Control : No

   Change which setting?
```

图 63.8.2.1 串口设置

设置好以后就可以静静的等待 GPS 数据输出，GPS 模块第一次启动可能需要几分钟搜星，等搜到卫星以后才会有定位数据输出。搜到卫星以后 GPS 模块输出的定位数据如图 63.8.2.2 所示：

```
Welcome to minicom 2.7.1

OPTIONS: I18n
Compiled on Sep 13 2019, 22:31:25.
Port /dev/ttymxc2, 16:27:59

Press CTRL-A Z for help on special keys
$GNGGA,034407.000,2318.2291,N,11319.5972,E,1,06,2.6,21.8,M,-5.4,M,,0000*61
$GNGLL,2318.2291,N,11319.5972,E,034407.000,A,A*41
$GPGSA,A,3,02,12,06,09,25,05,,,,,,,3.6,2.6,2.4*39
$GPGSV,3,1,10,19,54,116,01,05,51,249,15,02,44,334,21,06,41,040,28*76
$GPGSV,3,2,10,17,39,127,08,12,36,280,25,09,25,058,25,13,11,184,06*7F
$GPGSV,3,3,10,25,09,313,22,23,00,038,15*79
$GNRMC,034407.000,A,2318.2291,N,11319.5972,E,000.0,191.3,140919,,,A*78
$GNVTG,191.3,T,,M,000.0,N,000.0,K,A*19
$GNZDA,034407.000,14,09,2019,00,00*4A
$GNGGA,034408.000,2318.2291,N,11319.5972,E,1,06,3.9,21.8,M,-5.4,M,,0000*60
$GNGLL,2318.2291,N,11319.5972,E,034408.000,A,A*4E
$GPGSA,A,3,12,06,09,25,05,02,,,,,,,5.5,3.9,4.0*30
$GNRMC,034408.000,A,2318.2291,N,11319.5972,E,000.0,191.3,140919,,,A*77
$GNVTG,191.3,T,,M,000.0,N,000.0,K,A*19
$GNZDA,034410.000,14,09,2019,00,00*4C95972,E,000.0,191.3,140919,,,A*7E0*69
```

模块输出的GPS定位数据

图 63.8.2.2 GPS 数据

第六十四章 Linux 多点电容触摸屏实验

第六十五章 Linux 音频驱动实验

第六十六章 Linux CAN 驱动实验

第六十七章 Linux USB 驱动实验

第六十八章 Linux 块设备驱动实验

第六十九章 Linux 网络驱动实验

第七十章 Linux WIFI 驱动实验

WIFI 的使用已经很常见了，手机、平板、汽车等等，虽然可以使用有线网络，但是有时候很多设备存在布线困难的情况，此时 WIFI 就是一个不错的选择。正点原子 I.MX6U-ALPHA 开发板支持 USB 和 SDIO 这两种接口的 WIFI，本章我们就来学习一下如何在 I.MX6U-ALPHA 开发板上使用 USB 和 SDIO 这两种 WIFI。

70.1 WIFI 驱动添加与编译

正点原子的 I.MX6U-ALPHA 开发板目前支持两种接口的 WIFI：USB 和 SDIO，其中 USB WIFI 使用使用的芯片为 RTL8188EUS，SDIO 接口的 WIFI 使用芯片为 RTL8189FS，也叫做 RTL8189FTV。这两个都是 realtek 公司出品的 WIFI 芯片。WIFI 驱动不需要我们编写，因为 realtek 公司提供了 WIFI 驱动源码，因此我们只需要将 WIFI 驱动源码添加到 Linux 内核中，然后通过图形化界面配置，选择将其编译成模块即可。

正点原子 I.MX6U-ALPHA 开发板默认会赠送一个 RTL8188 USB WIFI，RTL8188 USB WIFI 如图 70.1.1 所示：



图 70.1.1 RTL8188 USB WIFI

另外，正点原子还有一款采用 RTL8189FTV 芯片的 SDIO WIFI，如图 70.1.2 所示：

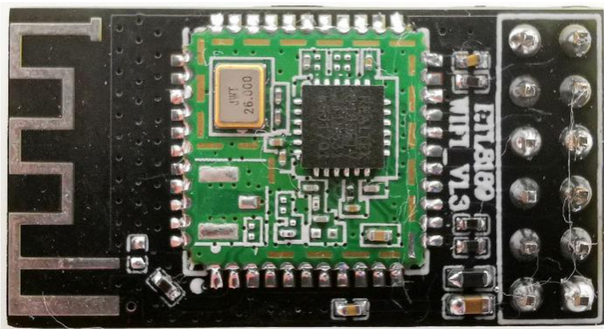


图 70.1.2 RTL8188 SDIO WIFI

70.1.1 向 Linux 内核添加 WIFI 驱动

1、rtl81xx 驱动文件浏览

WIFI 驱动源码已经放到了开发板光盘中，路径为：1、例程源码->5、模块驱动源码->1、RTL8XXX WIFI 驱动源码-> realtek。realtek 目录下就存放着 RTL8188EUS 和 RTL8189FS 这两个芯片的驱动源码，如图 70.1.1.1 所示：

名称	修改日期	类型	大小
rtl8188EUS	2019-09-14 16:59	文件夹	
rtl8189FS	2019-09-14 21:01	文件夹	
Kconfig	2019-09-14 19:13	文件	1 KB
Makefile	2019-09-14 16:54	文件	1 KB

图 70.1.1.1 rtl8xxx WIFI 驱动

其中 rtl8188EUS 下存放着 RTL8188EUS 驱动，RTL8189FS 存放着 RTL8189FS/FTV 的驱动文件。Kconfig 文件是 WIFI 驱动的配置界面文档，这样可以通过 Linux 内核图形化配置界面来选择是否编译 WIFI 驱动，Kconfig 文件内容如下所示：

示例代码 70.1.1.1 Kconfig 文件内容

```
1  menuconfig REALTEK_WIFI
```

```
2       tristate "Realtek wifi"
3
4   if REALTEK_WIFI
5
6   choice
7       prompt "select wifi type"
8       default RTL9189FS
9
10  config RTL9189FS
11      depends on REALTEK_WIFI
12      tristate "rtl8189fs/ftv sdio wifi"
13
14  config RTL8188EUS
15      depends on REALTEK_WIFI
16      tristate "rtl8188eus usb wifi"
17
18  endchoice
19  endif
```

Makefile 文件内容如下所示

示例代码 70.1.1.2 Makefile 文件内容

```
1 obj-$(CONFIG_RTL8188EUS) += rtl8188EUS/
2 obj-$(CONFIG_RTL8189FS) += rtl8189FS/
```

2、将 rtl81xx 驱动添加到 Linux 内核中

将 realtek 整个目录拷贝到 ubuntu 下 Linux 内核源码中的 drivers/net/wireless 目录下，此目录下存放着所有 WIFI 驱动文件。拷贝完成以后此目录如图 70.1.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless$ ls
adm8211.c        atmel_cs.c    hostap           Makefile          ray_cs.h       wl3501_cs.c
adm8211.h        atmel.h       ipw2x00          modules.builtin   rayctl.h       wl3501.h
airo.c           atmel_pci.c   iwlegacy         modules.order     realtek        zd1201.c
airo_cs.c        b43           iwlwifi          mwifiex           rndis_wlan.c   zd1201.h
airo.h           b43legacy     Kconfig          mwl8k.c           rsi            zd1211rw
at76c50x-usb.c   bcmdhd        libertas         orinoco           rt2x00
at76c50x-usb.h   brcm80211     libertas_tf      p54               rtl818x
ath              built-in.o    mac80211_hwsim.c prism54           rtlwifi
atmel.c          cw1200        mac80211_hwsim.h ray_cs.c          ti
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless$
```

图 70.1.1.1 拷贝完成的 wireless 目录

图 70.1.1.1 中框选出来的就是我们刚刚拷贝进来的 realtek 目录。

3、修改 drivers/net/wireless/Kconfig

打开 drivers/net/wireless/Kconfig，在里面加入下面这一行内容：

```
source "drivers/net/wireless/realtek/Kconfig"
```

添加完以后的 Kconfig 文件内容如下所示：

示例代码 70.1.1.3 drivers/net/wireless/Kconfig 文件内容

```
1 #
2 # Wireless LAN device configuration
```

```
3 #
4
5 menuconfig WLAN
......
286 source "drivers/net/wireless/rsi/Kconfig"
287 source "drivers/net/wireless/realtek/Kconfig"
286
289 endif # WLAN
```

第 287 行就是添加到 drivers/net/wireless/Kconfig 中的内容，这样 WIFI 驱动的配置界面才会出现在 Linux 内核配置界面上。

3、修改 drivers/net/wireless/Makefile

打开 drivers/net/wireless/Makefile，在里面加入下面一行内容：

```
obj-y += realtek/
```

修改完以后的 Makefile 文件内容如下所示：

示例代码 70.1.1.4 drivers/net/wireless/Makefile 文件内容

```
1 #
2 # Makefile for the Linux Wireless network device drivers.
3 #
4
5 obj-$(CONFIG_IPW2100) += ipw2x00/
......
62 obj-$(CONFIG_CW1200)    += cw1200/
63 obj-$(CONFIG_RSI_91X)   += rsi/
64
65 obj-y += realtek/
```

第 65 行，编译 realtek 中的内容，至此，Linux 内核要修改的内容就全部完成了。

70.1.2 配置 Linux 内核

在编译 RTL8188 和 RTL8189 驱动之前需要先配置 Linux 内核。

1、配置 USB 支持设备

配置路径如下：

```
-> Device Drivers
    -> <*> USB support
        -> <*> Support for Host-side USB
            -> <*> EHCI HCD (USB 2.0) support
            -> <*> OHCI HCD (USB 1.1) support
            -> <*> ChipIdea Highspeed Dual Role Controller
                -> [*] ChipIdea device controller
                -> [*] ChipIdea host controller
```

2、配置支持 WIFI 设备

配置路径如下：

-> Device Drivers
    -> [*] Network device support
        -> [*] Wireless LAN
            -> <*> IEEE 802.11 for Host AP (Prism2/2.5/3 and WEP/TKIP/CCMP)
                -> [*] Support downloading firmware images with Host AP driver
                -> [*] Support for non-volatile firmware download

配置完如图 70.1.2.1 所示：

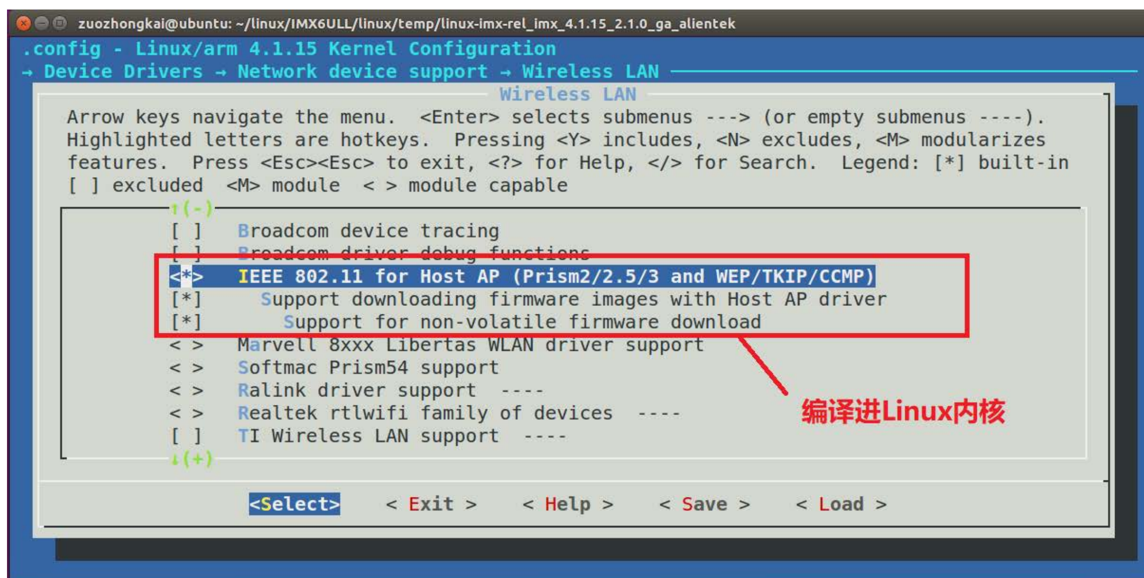


图 70.1.2.1 配置支持 WIFI 设备

3、配置支持 IEEE 802.11

配置路径如下：

-> Networking support
    -> -*- Wireless
        -> [*] cfg80211 wireless extensions compatibility
        -> <*> Generic IEEE 802.11 Networking Stack (mac80211)

配置完如图 70.1.2.2 所示：

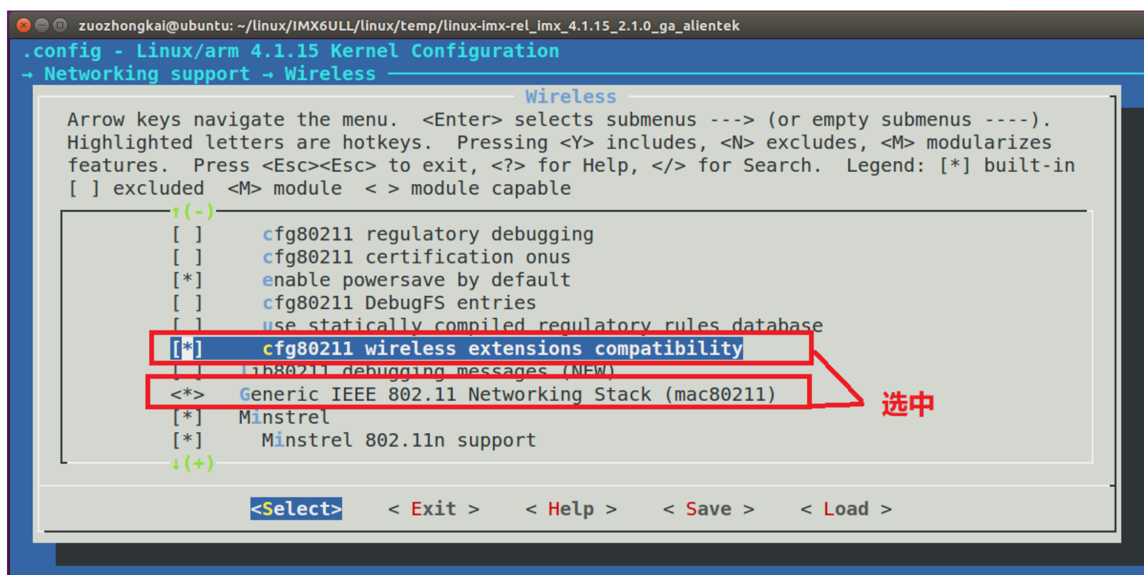


图 70.1.2.2 IEE 802.11 配置项

配置好以后重新编译一下 Linux 内核，得到新的 zImage，后面使用新编译出来的 zImage 启动系统。

70.1.3 编译 WIFI 驱动

执行“make menuconfig”命令，打开 Linux 内核配置界面，然后按照如下路径选择将 rtl81xx 驱动编译为模块：

```
-> Device Drivers
    -> Network device support (NETDEVICES [=y])
        -> Wireless LAN (WLAN [=y])
            -> Realtek wifi (REALTEK_WIFI [=m])
                -> rtl8189ftv sdio wifi
                -> rtl8188eus usb wifi
```

配置结果如图 70.1.3.1 所示：

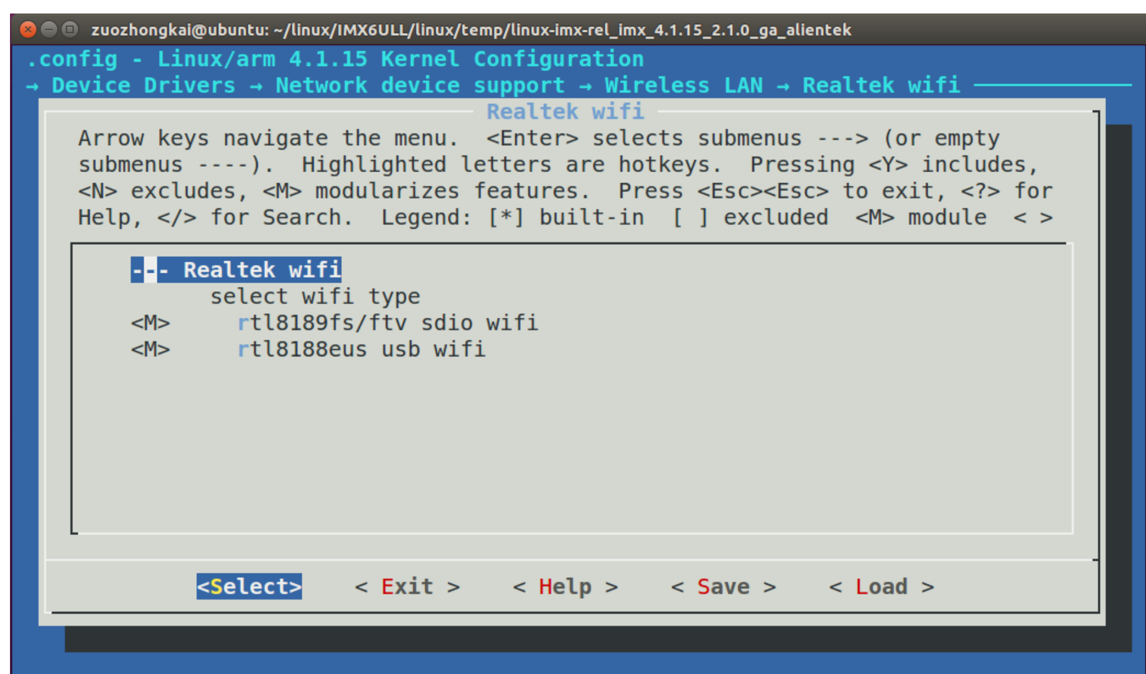


图 70.1.3.1 WIFI 配置界面

图 70.1.3.1 中的配置界面就是我们添加进去的 WIFI 配置界面，选中“rtl8189fs/ftv sdio wifi”和“rtl8188eus usb wifi”，将其编译为模块。执行如下命令编译模块：

```
make modules -j12        //编译驱动模块
```

编译完成以后就会在 rtl8188EUS 和 rtl8189FS 文件夹下分别生成 8188eu.ko 和 8189fs.ko 这两个.ko 文件，结果如图 70.1.3.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless/realtek/rtl8188EUS$ ls
8188eu.ko      8188eu.o  hal         Kconfig        os_dep    wlan0dhcp
8188eu.mod.c   clean     ifcfg-wlan0 Makefile       platform
8188eu.mod.o   core      include     modules.order  runwpa
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless/realtek/rtl8188EUS$ cd ../
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless/realtek$ cd rtl8189FS/
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless/realtek/rtl8189FS$ ls
8189fs.ko      8189fs.o  hal         Kconfig        os_dep    wlan0dhcp
8189fs.mod.c   clean     ifcfg-wlan0 Makefile       platform
8189fs.mod.o   core      include     modules.order  runwpa
zuozhongkai@ubuntu:~/linux/IMX6ULL/linux/temp/linux-imx-rel_imx_4.1.15_2.1.0_ga_alientek/drivers/net/wireless/realtek/rtl8189FS$
```

图 70.1.3.2 编译结果

图 70.1.3.2 中的 8188eu.ko 和 8189fs.ko 就是我们需要的 RTL8188EUS 和 RTL8189FS 的驱动模块文件，将这两个文件拷贝到 rootfs/lib/modules/4.1.15 目录中，命令如下：

```
sudo cp 8189fs.ko /home/zuozhongkai/linux/nfs/rootfs/lib/modules/4.1.15/ -rf
sudo cp 8188eu.ko /home/zuozhongkai/linux/nfs/rootfs/lib/modules/4.1.15/ -rf
```

因为我们重新配置过 Linux 内核，因此也需要使用新的 zImage 启动，将新编译出来的 zImage 镜像文件拷贝到 Ubuntu 中的 tftpboot 目录下，命令如下：

```
cp arch/arm/boot/zImage /home/zuozhongkai/linux/tftpboot/ -f
```

然后重启开发板！！！

70.1.4 驱动加载测试

1、RTL8188 USB WIFI 驱动测试

重启以后我们试着加载一下 8188eu.ko 和 8189fs.ko 这两个驱动文件，首先测试一下 RTL8188 的驱动文件，将 RTL8188 WIFI 模块插到开发板的 USB HOST 接口上。进入到目录 lib/modules/4.1.15 中，输入如下命令加载 8188eu.ko 这个驱动模块:

```
depmod                //第一次加载驱动的时候需要运行此命令
modprobe 8188eu.ko    //加载驱动模块
```

如果驱动加载成功的话如图 70.1.4.1 所示：

```
/lib/modules/4.1.15 # modprobe 8188eu.ko
RTL871X: module init start
RTL871X: rtl8188eu v4.3.0.9_15178.20150907
RTL871X: build time: Sep 14 2019 21:09:29
bFWReady == _FALSE call reset 8051...
RTL871X: rtw_ndev_init(wlan0)
usbcore: registered new interface driver rtl8188eu
RTL871X: module init ret=0
/lib/modules/4.1.15 #
```

图 70.1.4.1 RTL8188 驱动加载成功

输入“ifconfig -a”命令，查看 wlanX(X=0….n)网卡是否存在，一般都是 wlan0，除非板子上有多个 WIFI 模块在工作，结果如图 70.1.4.2 所示：

```
/lib/modules/4.1.15 # ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          inet addr:192.168.1.251  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::204:9fff:fe04:d235/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4281 errors:0 dropped:60 overruns:0 frame:0
          TX packets:1408 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4498909 (4.2 MiB)  TX bytes:225300 (220.0 KiB)

eth1      Link encap:Ethernet  HWaddr 00:04:9F:04:D2:35
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

wlan0     Link encap:Ethernet  HWaddr 00:13:EF:F1:0A:D7
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

WIFI对应的网卡

图 70.1.4.2 当前开发板所有网卡

从图 70.1.4.2 中可以看出，当前开发板有一个叫做“wlan0”的网卡，这个就是 RTL8188 对应的网卡。

2、RTL8189 SDIO WIFI 驱动测试

测试完 RTL8188 以后，再来测试一下 RTL8189 这个 SDIO WIFI，因为 I.MX6U-ALPHA 开发板的 SDIO WIFI 接口与 SD 卡公用一个 SDIO 接口。因此 SD 卡和 SDIO WIFI 只能二选其一，一次只能一个工作，所以测试 RTL8189 SDIO WIFI 的时候需要拔插 SD 卡。SDIO WIFI 接口原理图如图 70.1.4.3 所示：

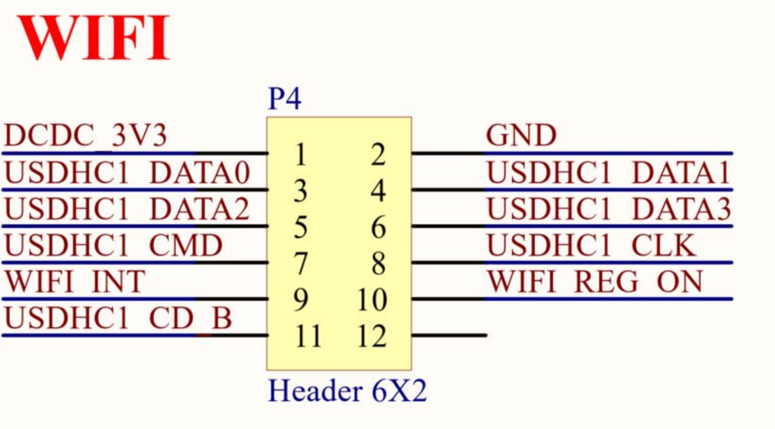


图 70.1.4.3 SDIO WIFI 接口

测试开始之前要先将 SD 卡拔出，然后将 RTL8189 SDIO WIFI 模块插入到 SDIO WIFI 座子上，如图 70.1.4.4 所示：

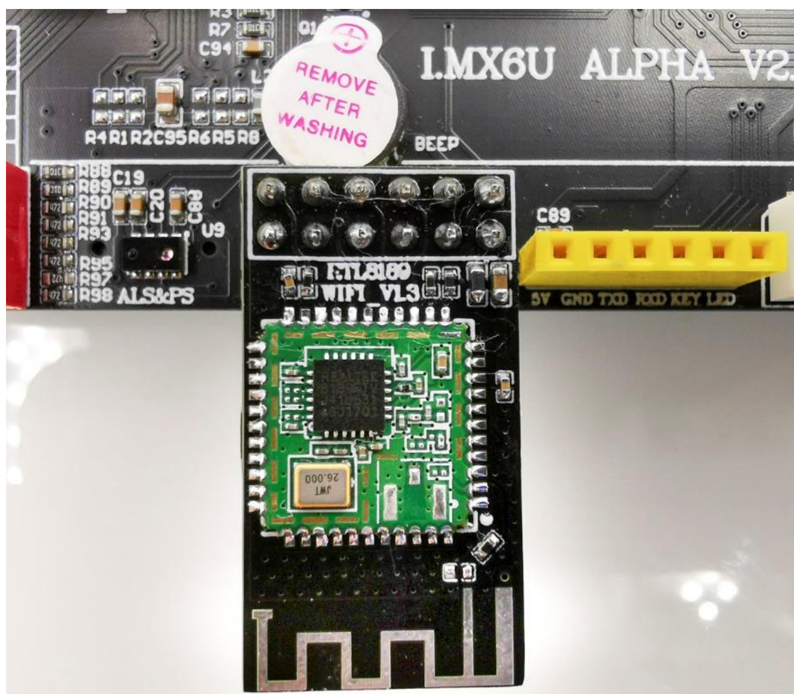


图 70.1.4.4 SDIO WIFI 连接图

SDIO WIFI 与开发板连接好以后就可以测试了，输入如下命令加载 8189fs.ko 这个驱动模块:

```
depmod                    //第一次加载驱动的时候需要运行此命令
modprobe 8189eu.ko        //加载驱动模块
```

如果驱动加载成功的话如图 70.1.4.5 所示：

```
/lib/modules/4.1.15 # modprobe 8189fs.ko
RTL871X: module init start
RTL871X: rtl8189fs v4.3.24.8_22657.20170607
RTL871X: HW EFUSE
RTL871X: hal_com_config_channel_plan chplan:0x20
RTL871X: rtw_regsty_chk_target_tx_power_valid return _FALSE for band:0, path:0, rs:0, t:-1
RTL871X: rtw_ndev_init(wlan0) if1 mac_addr=d4:b7:61:53:8f:e0
RTL871X: module init ret=0
/lib/modules/4.1.15 #
```

图 70.1.4.5 RTL8189 驱动加载成功

从 70.1.4.5 可以看出，RTL8189 SDIO WIFI 驱动加载成功，同样使用“ifconfig -a”命令查看一下是否有 wlanX(X=0…n)网卡存在，如果有的话就说明 RTL8189 SDIO WIFI 驱动工作正常。

不管是 RTL8188 USB WIFI 还是 RTL8189 SDIO WIFI，驱动测试都工作正常，但是我们得能联网啊，不能联网的话要他有什么用呢？WIFI 要想联网，需要移植一些其他第三方组件，否则无法连接路由器，接下来我们就移植这些第三方组件。

70.2 wireless tools 工具移植与测试

70.2.1 wireless tools 移植

wireless tools 是操作 WIFI 的工具集合，包括一下工具：

①、iwconfig：设置无线网络相关参数。

②、iwlist：扫描当前无线网络信息，获取 WIFI 热点。

③、iwspy：获取每个节点链接的质量。

④、iwpriv：操作 WirelessExtensions 特定驱动。

⑤、ifrename：基于各种静态标准命名接口。

我们最常用的就是 iwlist 和 iwconfig 这两个工具，首先获取到相应的源码包，这里我们已经放到了开发板光盘中，路径为：1、例程源码-》7、第三方库源码-》iwlist_for_visteon-master.tar.bz2。将 iwlist_for_visteon-master.tar.bz2 拷贝到 Ubuntu 中前面创建的 tool 目录下，拷贝完成以后将其解压，生成 iwlist_for_visteon-master 文件夹。进入到 iwlist_for_visteon-master 文件夹里面，打开 Makefile 文件，修改 Makefile 中的 CC、AR 和 RANLIB 这三个变量，修改后的值如图 70.2.1.1 所示：

```
10
11 ## Compiler to use (modify this for cross compile).
12 CC = arm-linux-gnueabihf-gcc
13 ## Other tools you need to modify for cross compile (static lib only).
14 AR = arm-linux-gnueabihf-ar
15 RANLIB = arm-linux-gnueabihf-ranlib
16
```

修改CC、AR和RANLIB

图 70.2.1.1 修改后的 CC、AR 和 RANLIB 值

图 70.2.1.1 中 CC、AR 和 RANLIB 这三个变量为所使用的编译器工具，将其改为我们所使用的 arm-linux-gnueabihf-xxx 工具即可。修改完成以后就可以使用如下命令编译：

```
make clean     //先清理一下工程
make           //编译
```

编译完成以后就会在当前目录下生成 iwlist、iwconfig、iwspy、iwpriv、ifrename 这 5 个工具，另外还有很重要的 libiw.so.29 这个库文件。将这 5 个工具拷贝到开发板根文件系统下的/usr/bin 目录中，将 libiw.so.29 这个库文件拷贝到开发板根文件系统下的/usr/lib 目录中，命令如下：

```
sudo cp iwlist iwconfig iwspy iwpriv ifrename /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
sudo cp libiw.so.29 /home/zuozhongkai/linux/nfs/rootfs/usr/lib/ -f
```

拷贝完成以后可以测试 iwlist 是否工作正常。

70.2.2 wireless tools 工具测试

这里我们主要测试一下 iwlist 工具，要测试 iwlist 工具，先测试一下 iwlist 工具能不能工作，输入 iwlist 命令，如果输出图 70.2.2.1 所示信息就表明 iwlist 工具工作正常。

```
/ # iwlist
Usage: iwlist [interface] scanning [essid NNN] [last]
              [interface] frequency
              [interface] channel
              [interface] bitrate
              [interface] rate
              [interface] encryption
              [interface] keys
              [interface] power
              [interface] txpower
              [interface] retry
              [interface] ap
              [interface] accesspoints
              [interface] peers
              [interface] event
              [interface] auth
              [interface] wpakeys
              [interface] genie
              [interface] modulation
/ #
```

图 70.2.2.1 iwlist 工具

正式测试 iwlist 之前得先让 WIFI 模块工作起来。RTL8188 或 RTL8189 都可以，以 RTL8188 USB WIFI 为例，先将 RTL8188 WIFI 模块插到开发板的 USB HOST 接口上，然后加载 RTL8188 驱动模块 8188eu.ko，驱动加载成功以后在打开 wlan0 网卡，命令如下：

```
modprobe 8188eu.ko          //加载 RTL8188 驱动模块
ifconfig wlan0 up           //打开 wlan0 网卡
```

wlan0 网卡打开以后就可以使用 iwlist 命令查找当前环境下的 WIFI 热点信息，也就是无线路由器，输入如下命令：

```
iwlist wlan0 scan
```

上述命令就会搜索当前环境下的所有 WIFI 热点，然后将这些热点的信息信息答应出来，包括 MAC 地址、ESSID(WIFI 名字)、频率、速率，信号质量等等，如图 70.2.2.2 所示：

```
/lib/modules/4.1.15 # iwlist wlan0 scan
wlan0     Scan completed :
          Cell 01 - Address: E4:0E:EE:F2:11:15
                    ESSID:""
                    Protocol:IEEE 802.11bgn
                    Mode:Master
                    Frequency:2.412 GHz (Channel 1)
                    Encryption key:on
                    Bit Rates:300 Mb/s
                    Extra:rsn_ie=30140100000fac040100000fac040100000fac020000
                    IE: IEEE 802.11i/WPA2 Version 1
                        Group Cipher : CCMP
                        Pairwise Ciphers (1) : CCMP
                        Authentication Suites (1) : PSK
                    Quality=51/100  Signal level=84/100
                    Extra:fm=0001
```

图 70.2.2.2 扫描到的 WIFI 热点信息

在扫描到的所有热点信息中找到自己要连接的 WIFI 热点，比如我要连接到“ZZK”这个热点上，这个 WIFI 热点信息如图 70.2.2.3 所示：

```
Cell 05 - Address: 88:F8:72:8D:F3:18
          ESSID:"ZZK"
          Protocol:IEEE 802.11bgn
          Mode:Master
          Frequency:2.437 GHz (Channel 6)
          Encryption key:on
          Bit Rates:300 Mb/s
          Extra:rsn_ie=30140100000fac040100000fac040100000fac020000
          IE: IEEE 802.11i/WPA2 Version 1
              Group Cipher : CCMP
              Pairwise Ciphers (1) : CCMP
              Authentication Suites (1) : PSK
          IE: Unknown: DD880050F204104A0001101044000102103B00010310470010630412531019203728DF31C1021000648756177656910230004575378781024000732303137 2D3131042000F31323334353633233343710540008000600 50F20400011011001141445 34C204D6F64656D2F526F7574657210080002068010 72A000120
          Quality=48/100  Signal level=100/100
          Extra:fm=0003
```

图 70.2.2.3 ZZK 热点信息

可以看出，“ZZK”这个热点信息已经被扫描到了，因此可以连接。要想连接到指定的 WIFI 热点上就需要用到 wpa_supplicant 工具，所以接下来就是移植此工具。

70.3 wpa_supplicant 移植

70.3.1 libopenssl 移植

wpa_supplicant 依赖于 libopenssl，因此需要先移植 libopenssl，libopenssl 源码已经放到了开发板光盘中，路径为：1、例程源码-》7、第三方库源码-》openssl-1.1.1-stable-SNAP-20190915.tar.gz。将 openssl 源码压缩包拷贝到 Ubuntu 中前面创建的 tool 目录下，然后使用如下命令将其解压：

```
tar -vxzf openssl-1.1.1-stable-SNAP-20190915.tar.gz
```

解压完成以后就会生成一个名为 openssl-1.1.1-stable-SNAP-20190915 的目录，然后在新建一个名为“libopenssl”的文件夹，用于存放 libopenssl 的编译结果。进入到解压出来的 openssl-1.1.1-stable-SNAP-20190915 目录中，然后执行如下命令进行配置：

```
./config shared no-asm --prefix=/home/zuozhongkai/linux/IMX6ULL/tool/libopenssl
```

配置成功以后会生成 Makefile，打开 Makefile，找到所有包含“-m64”的内容，一共两处分别为变量 CNF_CFLAGS 和 CNF_CXXFLAGS，将这两个变量中的“-m64”删除掉，删除以后如图 70.3.1.1 所示：

```
123 # Variables starting with CNF_ are common variables for all product types
124
125 CNF_CPPFLAGS=-DNDEBUG
126 #CNF_CFLAGS=-pthread -m64
127 #CNF_CXXFLAGS=-std=c++11 -pthread -m64
128 CNF_CFLAGS=-pthread
129 CNF_CXXFLAGS=-std=c++11 -pthread
```

将这两个变量中的"-m64"删除以后的新值

图 70.3.1.1 删除掉“-m64”

Makefile 修改好以后使用如下命令编译并安装 libopenssl：

```
make CROSS_COMPILE=arm-linux-gnueabihf- -j12
make install
```

编译安装完成以后的 libopenssl 目录内容如图 70.3.1.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libopenssl$ ls
bin  include  lib  share  ssl
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libopenssl$
```

图 70.3.1.2 编译并安装成功的 libopenssl 目录

将图 70.3.1.2 中的 lib 目录是我们需要的，将 lib 目录下的所有文件拷贝到开发板根文件系统中的/usr/lib 目录下，命令如下：

```
sudo cp lib/* /home/zuozhongkai/linux/nfs/rootfs/usr/lib/ -rf
```

70.3.2 libnl 库移植

wpa_supplicant 也依赖于 libnl，因此还需要移植一下 libnl 库，libnl 源码已经放到了开发板光盘中，路径为：1、例程源码-》7、第三方库源码-》libnl-3.2.23.tar.gz。将 libnl 源码压缩包拷贝到 Ubuntu 中前面创建的 tool 目录下，然后使用如下命令将其解压：

```
tar -vxzf libnl-3.2.23.tar.gz
```

得到解压完成以后会得到 libnl-3.2.23 文件夹，然后在新建一个名为"libnl"的文件夹，用于存放 libnl 的编译结果。进入到 libnl-3.2.23 文件夹中，然后执行如下命令进行配置：

```
./configure --host=arm-linux-gnueabihf --prefix=/home/zuozhongkai/linux/IMX6ULL/tool/libnl/
```

--host 用于指定交叉编译器的前缀，这里设置为"arm-linux-gnueabihf"，--prefix 用于指定编译结果存放目录，这里肯定要设置为我们刚刚创建的 libnl 文件夹。配置完成以后就可以执行如下命令对 libnl 库进行编译、安装：

```
make -j12          //编译
make install       //安装
```

编译安装完成以后的 libnl 目录如图 70.3.2.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl$ ls
etc  include  lib  sbin  share
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl$ 
```

图 70.3.2.1 编译安装完成后的 libnl 目录

我们需要图 70.3.2.1 中 lib 目录下的 libnl 库文件，将 lib 目录下的所有文件拷贝到开发板根文件系统的/usr/lib 目录下，命令如下所示：

```
sudo cp lib/* /home/zuozhongkai/linux/nfs/rootfs/usr/lib/ -rf
```

70.3.3 wpa_supplicant 移植

接下来移植 wpa_supplicant，wpa_supplicant 源码我们已经放到了开发板光盘中，路径为：1、例程源码->7、第三方库源码->wpa_supplicant-2.7.tar.gz，将 wpa_supplicant-2.7.tar.gz 拷贝到 Ubuntu 中，输入如下命令进行解压：

```
tar -vxzf wpa_supplicant-2.7.tar.gz
```

解压完成以后会得到 wpa_supplicant-2.7 文件夹，进入到此文件夹中，wpa_supplicant-2.7 目录内容如图 70.3.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool$ cd wpa_supplicant-2.7/
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/wpa_supplicant-2.7$ ls
CONTRIBUTIONS  COPYING  hs20  README  src  wpa_supplicant
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/wpa_supplicant-2.7$ 
```

图 70.3.3.1 wpa_supplicant-2.7 目录

进入到图 70.3.3.1 中的 wpa_supplicant 目录下，然后进行配置，wpa_supplicant 的配置比较特殊，需要将 wpa_supplicant 下的 defconfig 文件拷贝一份并重命名为.config，命令如下：

```
cd wpa_supplicant/
cp defconfig .config
```

完成以后打开.config 文件，在里面指定交叉编译器、openssl、libnl 库和头文件路径，设置如下：

示例代码 70.3.3.1 .config 文件需要添加的内容

```
1 CC = arm-linux-gnueabihf-gcc
```

```
2
3 #openssl 库和头文件路径
4 CFLAGS += -I/home/zuozhongkai/linux/IMX6ULL/tool/libopenssl/include
5 LIBS += -L/home/zuozhongkai/linux/IMX6ULL/tool/libopenssl/lib  -lssl
        -lcrypto
6
7 #libnl 库和头文件路径
8 CFLAGS += -I/home/zuozhongkai/linux/IMX6ULL/tool/libnl/include/libnl3
9 LIBS += -L/home/zuozhongkai/linux/IMX6ULL/tool/libnl/lib
```

CC 变量用于指定交叉编译器，这里就是 arm-linux-gnueabihf-gcc，CFLAGS 指定需要使用的库头文件路径，LIBS 指定需要用到的库路径。编译 wap_supplicant 的时候需要用到 openssl 和 libnl 库，所以示例代码 70.3.3.1 中指定了这两个的库路径和头文件路径。上述内容在.config 中的位置见图 70.3.3.2：

```
46 # Use libnl 3.2 libraries (if this is selected, CONFIG_LIBNL20 is ignored)
47 CONFIG_LIBNL32=y
48
49 CC = arm-linux-gnueabihf-gcc
50 CFLAGS += -I/home/zuozhongkai/linux/IMX6ULL/tool/libopenssl/include
51 LIBS += -L/home/zuozhongkai/linux/IMX6ULL/tool/libopenssl/lib  -lssl -lcrypto
52
53 CFLAGS += -I/home/zuozhongkai/linux/IMX6ULL/tool/libnl/include/libnl3
54 LIBS += -L/home/zuozhongkai/linux/IMX6ULL/tool/libnl/lib
```

添加进来的内容

图 70.3.3.2 添加到.config 中的内容

.config 文件配置好以后就可以编译 wpa_supplicant 了，使用如下命令编译：

```
export PKG_CONFIG_PATH=/home/zuozhongkai/linux/IMX6ULL/tool/libnl/lib/pkgconfig:
        $PKG_CONFIG_PATH            //指定 libnl 库 pkgconfig 包位置
make -j12                           //编译
```

首先我们使用 export 指定了 libnl 库的 pkgconfig 路径，环境变量 PKG_CONFIG_PATH 保存着 pkgconfig 包路径。在 tool/libnl/lib/下有个名为“pkgconfig”的目录，如图 70.3.3.3 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl/lib$ ls
libnl                 libnl-cli-3.so            libnl-idiag-3.a            libnl-nf-3.so.200
libnl-3.a             libnl-cli-3.so.200        libnl-idiag-3.la           libnl-nf-3.so.200.18.0
libnl-3.la            libnl-cli-3.so.200.18.0   libnl-idiag-3.so           libnl-route-3.a
libnl-3.so            libnl-genl-3.a            libnl-idiag-3.so.200       libnl-route-3.la
libnl-3.so.200        libnl-genl-3.la           libnl-idiag-3.so.200.18.0  libnl-route-3.so
libnl-3.so.200.18.0   libnl-genl-3.so           libnl-nf-3.a               libnl-route-3.so.200
libnl-cli-3.a         libnl-genl-3.so.200       libnl-nf-3.la              libnl-route-3.so.200.18.0
libnl-cli-3.la        libnl-genl-3.so.200.18.0  libnl-nf-3.so              pkgconfig
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl/lib$
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl/lib$
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/libnl/lib$
```

libnl下的pkgconfig，wpa_supplicant编译的时候需要用到

图 70.3.3.3 libnl 的 pkgconfig 目录

编译 wpa_supplicant 的时候是需要指定 libnl 的 pkgconfig 路径，否则会提示“libnl-3.0”或者“libnl-3.0.pc”找不到等错误。编译完成以后就会在本目录下生成 wpa_supplicant 和 wpa_cli 这两个软件，如图 70.3.3.3 所示：

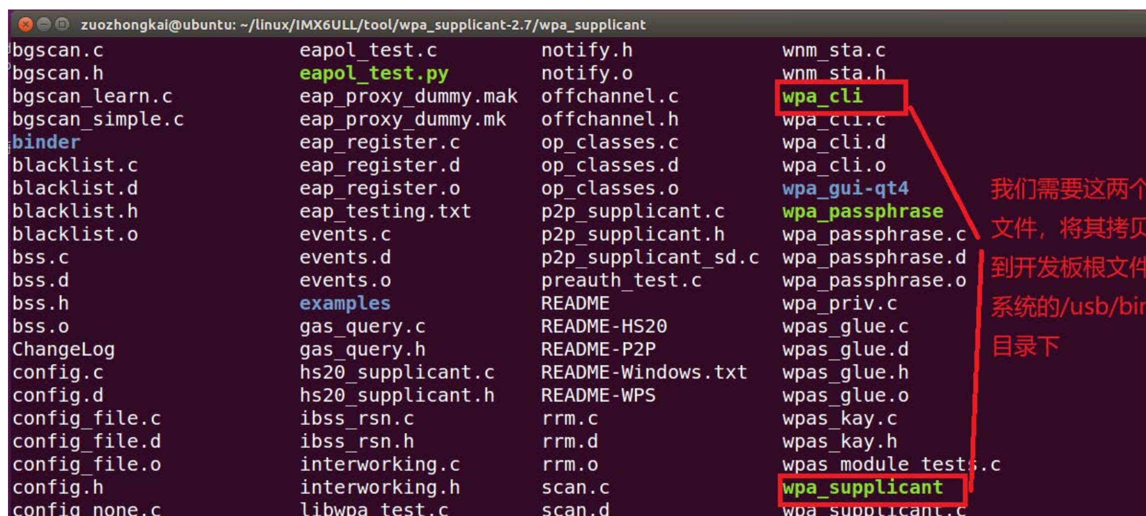


图 70.3.3.3 编译出来的 wpa_cli 和 wpa_supplicant 文件

将图 70.3.3.3 中的 wpa_cli 和 wpa_supplicant 这两个文件拷贝到开发板根文件系统的/usr/bin 目录中，命令如下：

```
sudo cp wpa_cli wpa_supplicant /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
```

拷贝完成以后重启开发板！输入"wpa_supplicant -v"命令查看一下 wpa_supplicant 版本号，如果 wpa_supplicant 工作正常的话就会打印出版本号，如图 70.3.3.4 所示：

```
/ # wpa_supplicant  -v
wpa_supplicant v2.7
Copyright (c) 2003-2018, Jouni Malinen <j@w1.fi> and contributors
/ # 
```

图 70.3.3.4 wpa_supplicant 版本号

从图 70.3.3.4 可以看出， wpa_supplicant 的版本号输出正常，说明 wpa_supplicant 移植成功，接下来就是使用 wpa_supplicant 将开发板的 WIFI 链接到路由器上，实现 WIFI 上网功能。

70.4 WIFI 联网测试

不管是 USB WIFI 还是 SDIO WIFI，联网的操作步骤如下所示：

①、插上 WIFI 模块，如果是板子集成的就不需要这一步。如果是 SDIO WIFI 的话确保 WIFI 所使用的 SDIO 接口没有插其他的模块，比如 SD 卡，防止其他模块对 SDIO WIFI 造成影响。

②、加载 RTL8188 或者 RTL8189 驱动模块。

③、使用 ifconfig 命令打开对应的无线网卡，比如 wlan0 或 wlan1……

④、无线网卡打开以后使用 iwlist 命令扫描一下当前环境下的 WIFI 热点，一来测试一下 WIFI 工作是否正常。二来检查一下自己要连接的 WIFI 热点能不能扫描到，扫描不到的话肯定就没法连接了。

当上述步骤确认无误以后就可以使用 wpa_supplicant 来将 WIFI 连接到指定的热点上，实现联网功能。

70.4.1 RTL8188 USB WIFI 联网测试

首先测试一下 RTL8188 USB WIFI 联网测试，确保 RTL8188 能扫描出要连接的 WIFI 热点，比如我要连接"ZZK"这个 WIFI，iwlist 扫描到的此 WIFI 热点信息如图 70.4.1.1 所示：

```
Cell 02 - Address: 88:F8:72:8D:F3:18
          ESSID:"ZZK"
          Protocol:IEEE 802.11bgn
          Mode:Master
          Frequency:2.437 GHz (Channel 6)
          Encryption key:on
          Bit Rates:300 Mb/s
          Extra:rsn_ie=30140100000fac040100000fac040100000fac020000
          IE: IEEE 802.11i/WPA2 Version 1
              Group Cipher : CCMP
              Pairwise Ciphers (1) : CCMP
              Authentication Suites (1) : PSK
```

图 70.4.1.1 ZZK WIFI 热点

要连接的 WIFI 热点扫描到以后就可以连接了，先在开发板根文件系统的/etc 目录下创建一个名为“wpa_supplicant.conf”的配置文件，此文件用于配置要连接的 WIFI 热点以及 WIFI 秘密，比如我要连接到“ZZK”这个热点上，因此 wpa_supplicant.conf 文件内容如下所示：

示例代码 70.4.1.1 wpa_supplicant.conf 文件内容

```
1 ctrl_interface=/var/run/wpa_supplicant
2 ap_scan=1
3 network={
4  ssid="ZZK"
5  psk="xxxxxxxx"
6 }
```

第 4 行，ssid 是要连接的 WIFI 热点名字，这里我要连接的是“ZZK”这个 WIFI 热点。

第 5 行，psk 就是要连接的 WIFI 热点密码，根据自己的实际情况填写即可。

注意，wpa_supplicant.conf 文件对于格式要求比较严格，“=”前后一定不能有空格，也不要用 TAB 键来缩进，比如第 4 行和 5 行的缩进应该采用空格，否则的话会出现 wpa_supplicant.conf 文件解析错误！最重要的一点！wpa_supplicant.conf 文件内容要自己手动输入，不要偷懒复制粘贴！！！

wpa_supplicant.conf 文件编写好以后再在开发板根文件系统下创建一个“/var/run/wpa_supplicant”目录，wpa_supplicant 工具要用到此目录！命令如下：

```
mkdir /var/run/wpa_supplicant -p
```

一切准备好以后就可以使用 wpa_supplicant 工具让 RTL8188 USB WIFI 连接到热点上，输入如下命令：

```
wpa_supplicant -D wext -c /etc/wpa_supplicant.conf -i wlan0 &
```

当 RTL8188 连接上 WIFI 热点以后会输出如图 70.4.1.2 所示的信息：

```
RTL871X: rtw_aes_decrypt(wlan0) no_gkey_bc_cnt:4, no_gkey_mc_cnt:0
RTL871X: recv eapol packet
RTL871X: send eapol packet
RTL871X: set pairwise key camid:4, addr:88:f8:72:8d:f3:18, kid:0, type:AES
wlan0: WPA: Key negotiation completed with 88:f8:72:8d:f3:18 [PTKRTL871X: set group key camid:5, addr:
88:f8:72:8d:f3:18, kid:1, type:AES
=CCMP GTK=CCMP]
wlan0: CTRL-EVENT-CONNECTED - Connection to 88:f8:72:8d:f3:18 completed [id=0 id_str=]
```

连接成功

图 70.4.1.2 连接成功

从图 70.4.1.2 可以看出，当 RTL8188 连接到 WIFI 热点上以后会输出“wlan0: CTRL-EVENT-CONNECTED”字样。接下来就是最后一步了，设置 wlan0 的 IP 地址，这里使用 udhcpc 命令从路由器申请 IP 地址，输入如下命令：

```
udhcpc -i wlan0                //从路由器获取 IP 地址
```

IP 地址获取成功以后会输出如图 70.4.2.2 所示信息：

```
/lib/modules/4.1.15 # udhcpc -i wlan0
udhcpc: started, v1.29.0
Setting IP address 0.0.0.0 on wlan0
udhcpc: sending discover
udhcpc: sending select for 192.168.1.126
udhcpc: lease of 192.168.1.126 obtained, lease time 86400
Setting IP address 192.168.1.126 on wlan0
Deleting routers
route: SIOCDELRT: No such process
Adding router 192.168.1.1
Recreating /etc/resolv.conf
 Adding DNS server 114.114.114.114
/lib/modules/4.1.15 #
```

图 70.4.2.2 wlan0 网卡 WIFI 地址获取成功

从图 70.4.2.2 可以看出，wlan0 的 IP 地址获取成功，IP 地址为 192.168.1.126。可以输入如下命令查看一下 wlan0 网卡的详细信息：

```
ifconfig wlan0
```

结果如图 70.4.2.3 所示：

```
/lib/modules/4.1.15 # ifconfig wlan0
wlan0     Link encap:Ethernet  HWaddr 00:13:EF:F1:0A:D7
          inet addr:192.168.1.126  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::213:efff:fef1:ad7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2538 errors:0 dropped:127 overruns:0 frame:0
          TX packets:11 errors:0 dropped:3 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:262646 (256.4 KiB)  TX bytes:1783 (1.7 KiB)

/lib/modules/4.1.15 #
```

图 70.4.2.3 wlan0 网卡详细信息

可以通过电脑 ping 一下 wlan0 的 192.168.1.126 这个 IP 地址，如果能 ping 通就说明 RTL8188 USB WIFI 工作正常。也可以直接在开发板上使用 wlan0 来 ping 一下百度网站，输入如下命令：

```
ping -I 192.168.1.126 www.baidu.com
```

-I 是指定执行 ping 操作的网卡 IP 地址，我们要使用 wlan0 去 ping 百度网站，因此要通过"-I"指定 wlan0 的 IP 地址。如果 WIFI 工作正常的话就可以 ping 通百度网站，如图 70.4.2.4 所示：

```
/lib/modules/4.1.15 # ping -I 192.168.1.126 www.baidu.com
PING www.baidu.com (14.215.177.39) from 192.168.1.126: 56 data bytes
64 bytes from 14.215.177.39: seq=0 ttl=56 time=15.529 ms
64 bytes from 14.215.177.39: seq=1 ttl=56 time=25.117 ms
64 bytes from 14.215.177.39: seq=2 ttl=56 time=14.481 ms
64 bytes from 14.215.177.39: seq=3 ttl=56 time=16.314 ms
64 bytes from 14.215.177.39: seq=4 ttl=56 time=12.875 ms
```

图 70.4.2.4 百度网站 ping 成功

至此 RTL8188 USB WIFI 我们就完全驱动起来了，大家就可以使用 WIFI 来进行网络通信了。

70.4.2 RTL8189 SDIO WIFI 联网测试

RLT8189 SDIO WIFI 的测试和 RTL8188 USB WIFI 的测试方法基本一致，如果插了 SD 卡的话先将 SD 卡从 I.MX6U-ALPHA 开发板上拔出，因为 I.MX6U-ALPHA 开发板的 SD 卡和 SDIO WIFI 公用一个 SDIO 接口。插入 RTL8189 SDIO WIFI 模块，然后加载 RTL8189 驱动，并且打开对应的 wlan0(如果只有 RTL8189 一个 WIFI 的话)网卡，使用 iwlist 命令搜索要连接的 WIFI 热点是否存在，如果存在的话就可以连接了。

RTL8189 SDIO WIFI 同样使用 wpa_supplicant 来完成热点连接工作，因此同样需要创建 /etc/wpa_supplicant.conf 文件，具体过程参考 70.4.1 小节。一切准备就绪以后输入如下命令来完成 WIFI 热点连接：

```
wpa_supplicant -Dnl80211 -c /etc/wpa_supplicant.conf -i wlan0 &
```

注意红色字体，使用 RTL8189 的话应该使用“-Dnl80211”，这里不要填错了！WIFI 热点连接成功以后会输出如图 70.4.2.1 所示信息：

```
RTL871X: recv eapol packet
RTL871X: send eapol packet
RTL871X: set pairwise key camid:4, addr:88:f8:72:8d:f3:18, kid:0, type:AES
wlan0: WPA: Key negotiation complRTL871X: set group key camid:5, addr:88:f8:72:8d:f3:18, kid:
ES
eted with 88:f8:72:8d:f3:18 [PTK=CCMP GTK=CCMP]
wlan0: CTRL-EVENT-CONNECTED - Connection to 88:f8:72:8d:f3:18 completed [id=0 id_str=]

/lib/modules/4.1.15 #
```

WIFI连接成功

图 70.4.2.1 RTL8189 SDIO WIFI 连接成功

使用 udhcpc 命令获取 IP 地址，命令如下：

```
udhcpc -i wlan0
```

IP 地址获取过程如图 70.4.2.2 所示：

```
/lib/modules/4.1.15 # udhcpc -i wlan0
udhcpc: started, v1.29.0
Setting IP address 0.0.0.0 on wlan0
udhcpc: sending discover
udhcpc: sending select for 192.168.1.118
udhcpc: lease of 192.168.1.118 obtained, lease time 86400
Setting IP address 192.168.1.118 on wlan0
Deleting routers
route: SIOCDELRT: No such process
Adding router 192.168.1.1
Recreating /etc/resolv.conf
 Adding DNS server 114.114.114.114
/lib/modules/4.1.15 #
```

图 70.4.2.2 udhcpc 获取 IP 地址过程

从图 70.4.2.2 可以看出，wlan0 的 IP 地址为 192.168.1.118，大家可以使用“ifconfig wlan0”查看一下 wlan0 网卡的详细信息。可以通过电脑 ping 一下 192.168.1.118 测试 WIFI 是否工作正常，或者在开发板上使用 wlan0 网卡 ping 一下百度网址来测试一下 WIFI 工作是否正常，输入如下命令：

```
ping -I 192.168.1.118 www.baidu.com
```

如果 ping 成功的话结果如图 70.4.2.3 所示：

```
/lib/modules/4.1.15 # ping -I 192.168.1.118 www.baidu.com
PING www.baidu.com (14.215.177.39) from 192.168.1.118: 56 data bytes
64 bytes from 14.215.177.39: seq=0 ttl=56 time=11.410 ms
64 bytes from 14.215.177.39: seq=1 ttl=56 time=15.643 ms
64 bytes from 14.215.177.39: seq=2 ttl=56 time=21.756 ms
```

图 70.4.2.3 ping 百度网站测试成功

至此，如何在 I.MX6U-ALPHA 开发板上使用 WIFI 就全部讲解完了，包括 USB WIFI 和 SDIO WIFI。其实不管是在 I.MX6U 上，还是在其他的 SOC 上，USB WIFI 和 SDIO WIFI 的驱动都是类似的，大家可以参考本章教程讲 RTL8188、RTL8189 这两款 WIFI 的驱动移植到芯片或者开发板上。

第七十一章 Linux 4G 通信实验

前面我们学习了如何在 Linux 中使用有线网络或者 WIFI，但是使用有线网络或者 WIFI 有很多限制，因为要布线，即使是 WIFI 你也得先布线，然后在接个路由器。有很多场合是不方便布线的，这个时候就是 4G 大显身手的时候，产品可以直接通过 4G 连接到网络，实现无人值守。本章我们就来学一下如何在 I.MX6U-ALPHA 开发板中使用 4G 来实现联网功能。

71.1 4G 网络连接简介

提起 4G 网络连接，大家可能会觉得是个很难的东西，其实对于嵌入式 Linux 而言，4G 网络连接恰恰相反，不难！大家可以看一下其他的嵌入式 Linux 或者 Android 开发板，4G 模块都是 MiniPCIE 接口的，包括很多 4G 模块都是 MiniPCIE 接口的。但是大家稍微深入研究一下就会发现，这些 4G 模块虽然用了 MiniPCIE 接口，但是实际上的通信接口都是 USB，所以 4G 模块的驱动就转换为了 USB 驱动。而这些 4G 模块厂商都提供了详细的文档讲解如何在 Linux 下使用 4G 模块，以及如何修改 Linux 内核加入 4G 模块驱动。正点原子的 I.MX6U-ALPHA 开发板也有一个 MiniPCIE 形式的 4G 模块接口，虽然外形是 MiniPCIE 的，但是内心却是 USB 的。I.MX6U-ALPHA 开发板的 4G 模块原理图如图 71.4.1 所示：

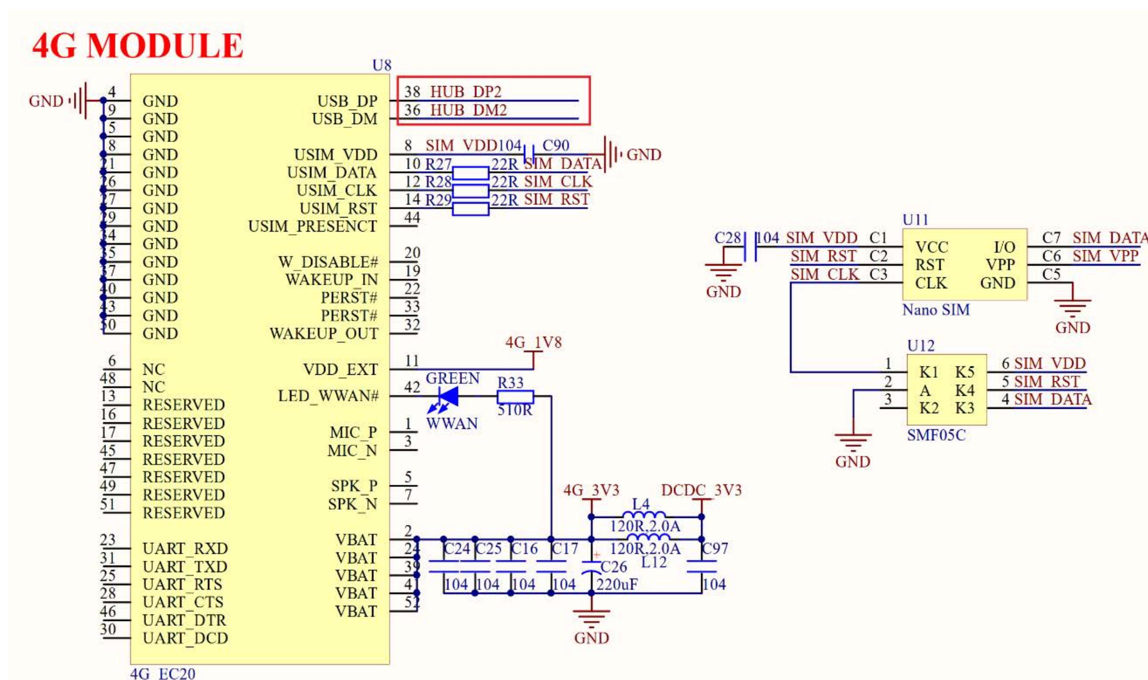


图 71.4.1 4G 模块原理图

图 71.4.1 中的 U8 就是 MiniPCIE 接口，MiniPCIE 接口连接到了 GL850 这个 HUB 芯片的 DP2 和 DM2，也就是 GL850 的 USB2 接口上。U11 就是 Nano SIM 接口，I.MX6U-ALPHA 开发板使用 Nano SIM 卡，这样大家就可以直接拿自己的手机卡进行测试。I.MX6U-ALPHA 开发板的 4G 接口位置如图 71.1.2 所示：

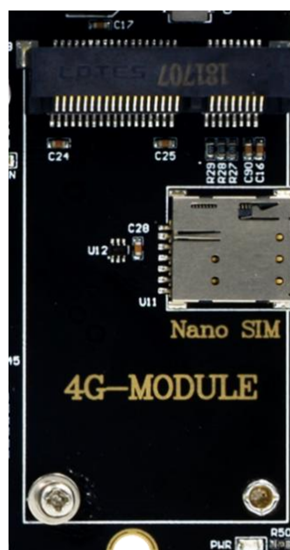


图 71.1.2 ALPHA 开发板 4G 接口

在使用之前需要将 4G 模块插入到图 71.1.2 所示的 MiniPCIE 接口上，然后上紧两边的螺丝，Nano SIM 卡插入到图 71.1.2 中的 Nano SIM 座里面，注意 Nano SIM 卡的方向！

Nano SIM 卡的金属触点朝下！

Nano SIM 卡的金属触点朝下！

Nano SIM 卡的金属触点朝下！

理论上所有的 MiniPCIE 接口的 4G 模块都可以连接到正点原子的 I.MX6U-ALPHA 开发板上，因为这些 4G 模块都遵循同样的接口标准，但是大家在使用的时候还是要详细的看一下 4G 模块的接口引脚描述。不同的 4G 模块其驱动形式也不同，本章我们讲解两款 4G 模块在 I.MX6U-ALPHA 开发板上的使用，一个是上海移远公司的 EC20，另外一个是高新兴物联的 ME3630，这两款 4G 模块都有 MiniPCIE 接口的，这两个 4G 模块如图 71.4.2 所示：



图 71.4.2 4G 模块

图 71.4.2 中左侧的是高新兴物联的 ME3630-W 4G 模块，右侧的是上海移远的 EC20 4G 模块。本章我们就分别来讲解一下如何在 I.MX6U-ALPHA 开发板上使用 EC20 和 ME3630 这两个 4G 模块。

4G 模块工作是需要天线的，因此在选购 4G 模块的时候一定要记得购买天线，否则无法进行测试。一般 MiniPCIE 接口的 4G 模块留出来的天线接口为 IPEX 座，因此购买天线的时候也要选择 IPEX 接口的，或者使用 IPEX 转 SMA 线来转接。推荐大家到正电原子官方店铺购买 4G 模块和相应的天线。

71.2 高新兴 ME3630 4G 模块实验

71.2.1 ME3630 4G 模块简介

ME3630 4G 模块是正点原子官方推荐的 4G 通信模块，ME3630 4G 模块是深圳高新兴物联出品的 4G LTE 模块，前身是中兴物联，正点原子是高新兴物联官方代理商，因此在模块质量以及售后服务这一块是有保证的。

ME3630 是一款 LTE Cat.4 七模全网通 4G 模块，在 LTE 模式下可以提供 50Mbps 上行速率以及 150Mbps 的下行速率，并支持回退到 3G 或 2G 网络。此模组支持分集接收、分集接收是终端产品支持双天线以提高通信质量和通信可靠性的无线连接技术。ME3630 支持多种网络协议，比如 PAP、CHAP、PPP 等，拥有多种功能，比如 GNSS、Remote wakeup、SMS、支持 FoTA 空中升级等。ME3630 4G 模块广泛应用于智能抄表、安防信息采集、工业路由器、车载通信以及监控等等 M2M 领域。ME3630 4G 模组特性如下：

①、一路 USB2.0 接口。
②、一路 UART 接口。
③、SIM 卡接口支持 1.8/3.0V。
④、内置 TCP、UDP、FTP 和 HTTP 等协议。
⑤、支持 RAS/ECM/NDIS。
⑥、支持 AT 指令。

ME3630 4G 模块有多种配置，比如纯数据版本、集成 GNSS 版本、全网通版本等等，本节教程我们主要使用到 ME3630 的数据通信功能，因此推荐大家购买全网通数据版，如果想要定位功能的话就购买全网通数据+GNSS 版本，至于其他的版本大家根据自己的实际需求选择即可。在正式使用 ME3630 4G 模块之前，请先将其插入到开发板的 MiniPCIE 座上、上紧螺、插入 Nano SIM 卡、接上天线，如图 71.2.1.1 所示：

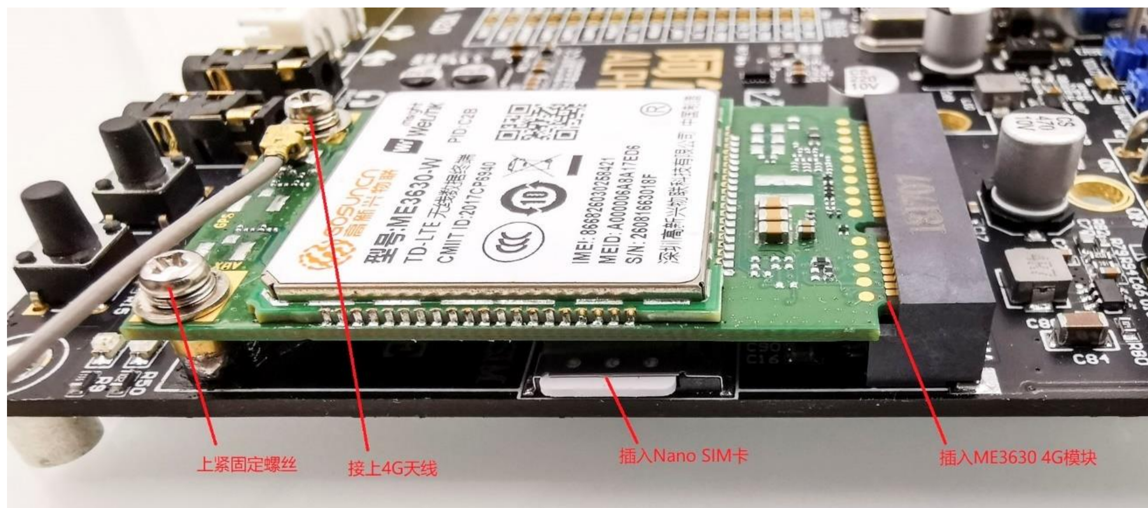


图 71.2.1.1 ALPHA 开发板连接 ME3630 模块

一切准备就绪以后就可以开始驱动 ME3630 4G 模块了。

71.2.2 ME3630 4G 模块驱动修改

1、添加 USB 设备信息

我们需要先在 Linux 内核中添加 ME3630 的 USB 设备信息，因为我们前面说了，ME3630 4G 模块用的 USB 接口。打开 Linux 源码的 drivers/usb/serial/option.c 文件，找到 options_ids 数组，然后在里面添加 ME3630 的 PID 和 VID， 要添加的内容如下：

示例代码 71.2.2.1 ME3630 PID 和 VID 信息

```
1 { USB_DEVICE(0x19d2, 0x0117) }, /* ME3630*/
2 { USB_DEVICE(0x19d2, 0x0199) },
3 { USB_DEVICE(0x19d2, 0x1476) },
```

完成以后的 options_ids 数组如图 71.2.2.1 所示：

```
630 static const struct usb_device_id option_ids[] = {
631     { USB_DEVICE(0x19d2, 0x0117) }, /* ME3630*/
632     { USB_DEVICE(0x19d2, 0x0199) },
633     { USB_DEVICE(0x19d2, 0x1476) },
634     { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_COLT) },
```

ME3630的PID和VID信息

图 71.2.2.1 添加 PID 和 VID 后的 option_ids 数组

2、添加 ECM 支持程序

ME3630 支持 ECM 接口，可以通过 ECM 接口轻松联网，如果要使用 ECM 接口的话需要修改 drivers/usb/serial/option.c 文件里面的 option_probe 函数。找到此函数，然后在里面输入如下内容：

示例代码 71.2.2.2 option_probe 函数需要添加的内容

```
1 /* EM3630 */
2 if (serial->dev->descriptor.idVendor == 0x19d2 &&
3     serial->dev->descriptor.idProduct == 0x1476 &&
4     serial->interface->cur_altsetting->desc. bInterfaceNumber == 3)
5     return -ENODEV;
```

```
6
7  if (serial->dev->descriptor.idVendor == 0x19d2 &&
8      serial->dev->descriptor.idProduct == 0x1476 &&
9      serial->interface->cur_altsetting->desc. bInterfaceNumber == 4)
10     return -ENODEV;
11
12 if (serial->dev->descriptor.idVendor == 0x19d2 &&
13     serial->dev->descriptor.idProduct == 0x1509 &&
14     serial->interface->cur_altsetting->desc. bInterfaceNumber == 4)
15     return -ENODEV;
16
17 if (serial->dev->descriptor.idVendor == 0x19d2 &&
18     serial->dev->descriptor.idProduct == 0x1509 &&
19     serial->interface->cur_altsetting->desc. bInterfaceNumber == 5)
20     return -ENODEV;
```

添加完成以后的 option_probe 函数如图 71.2.2.2 所示：

```
1889    if (dev_desc->idVendor == cpu_to_le16(SAMSUNG_VENDOR_ID) &&
1890        dev_desc->idProduct == cpu_to_le16(SAMSUNG_PRODUCT_GT_B3730) &&
1891        iface_desc->bInterfaceClass != USB_CLASS_CDC_DATA)
1892        return -ENODEV;
1893
1894    /* EM3630 */
1895    if (serial->dev->descriptor.idVendor == 0x19d2 &&
1896        serial->dev->descriptor.idProduct == 0x1476 &&
1897        serial->interface->cur_altsetting->desc. bInterfaceNumber == 3)
1898        return -ENODEV;
1899
1900    if (serial->dev->descriptor.idVendor == 0x19d2 &&
1901        serial->dev->descriptor.idProduct == 0x1476 &&
1902        serial->interface->cur_altsetting->desc. bInterfaceNumber == 4)
1903        return -ENODEV;
1904
1905    if (serial->dev->descriptor.idVendor == 0x19d2 &&
1906        serial->dev->descriptor.idProduct == 0x1509 &&
1907        serial->interface->cur_altsetting->desc. bInterfaceNumber == 4)
1908        return -ENODEV;
1909
1910    if (serial->dev->descriptor.idVendor == 0x19d2 &&
1911        serial->dev->descriptor.idProduct == 0x1509 &&
1912        serial->interface->cur_altsetting->desc. bInterfaceNumber == 5)
1913        return -ENODEV;
```

需要添加的内容

图 71.2.2.2 向 option_probe 添加的内容

3、配置 Linux 内核

我们需要配置 Linux 内核，首先使能 USBNET 功能，路径如下：

```
-> Device Drivers
    -> -*- Network device support
        -> USB Network Adapters
            -> -*- Multi-purpose USB Networking Framework
```

配置如图 71.2.2.3 所示：

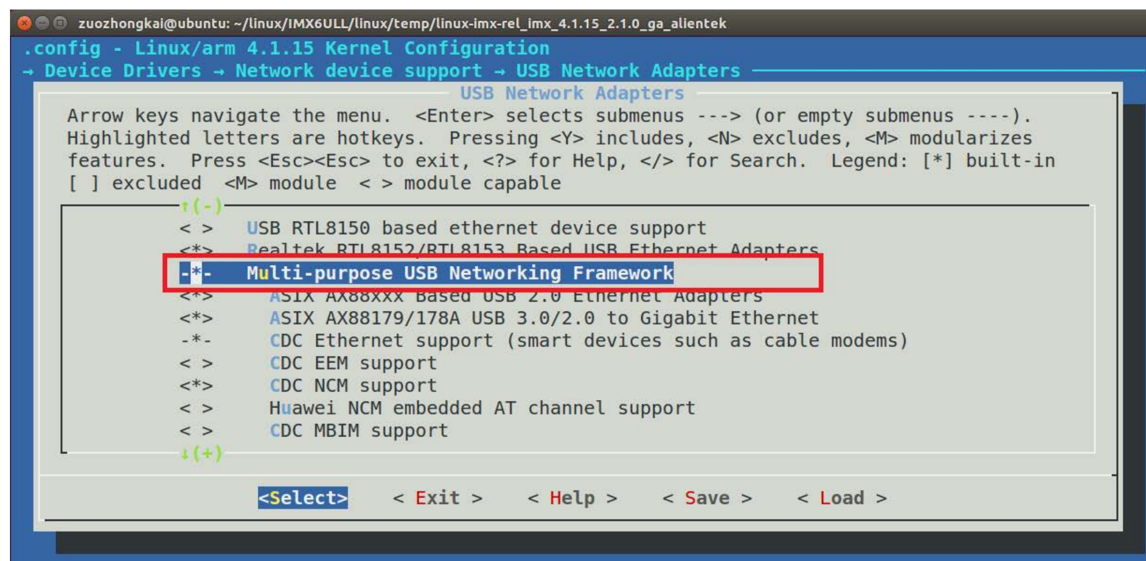


图 71.2.2.3 使能 USB 网络

接下来我们还需要使能 USB 串口 GSM、CDMA 驱动，配置路径如下：

```
-> Device Drivers
    -> [*] USB support
        -> <*> USB Serial Converter support
            -> <*> USB driver for GSM and CDMA modems
```

配置如图 71.2.2.4 所示：

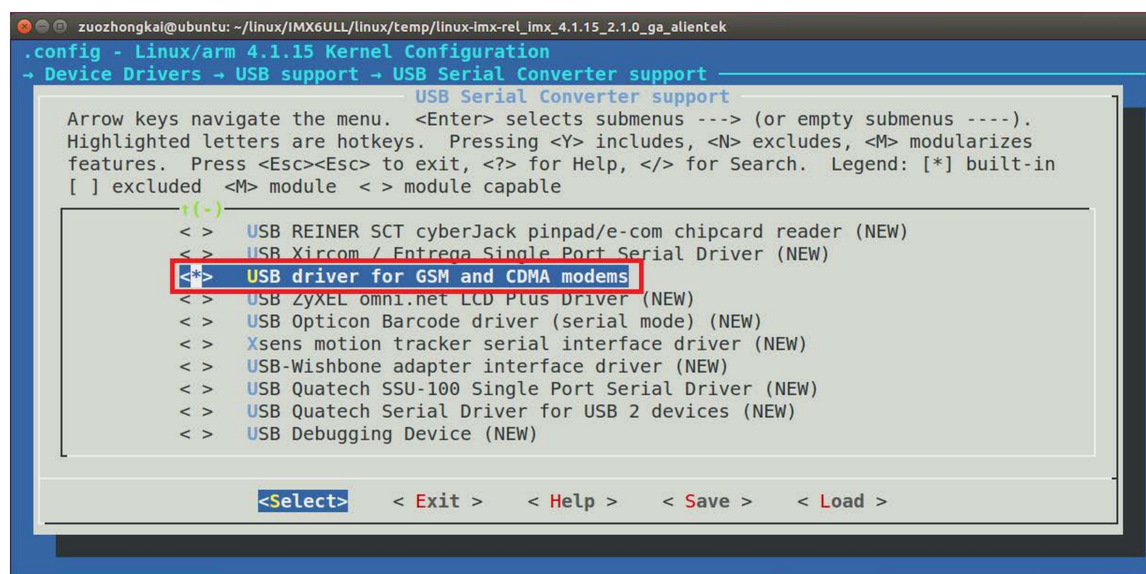


图 71.2.2.4 USB GSM 和 CDMA

继续配置 Linux 内核，使能 USB 的 CDC ACM 模式，配置路径如下：

```
-> Device Drivers
    -> [*] USB support
        -> <*> Support for Host-side USB
            -> <*> USB Modem (CDC ACM) support
```

配置如图 71.2.2.5 所示：

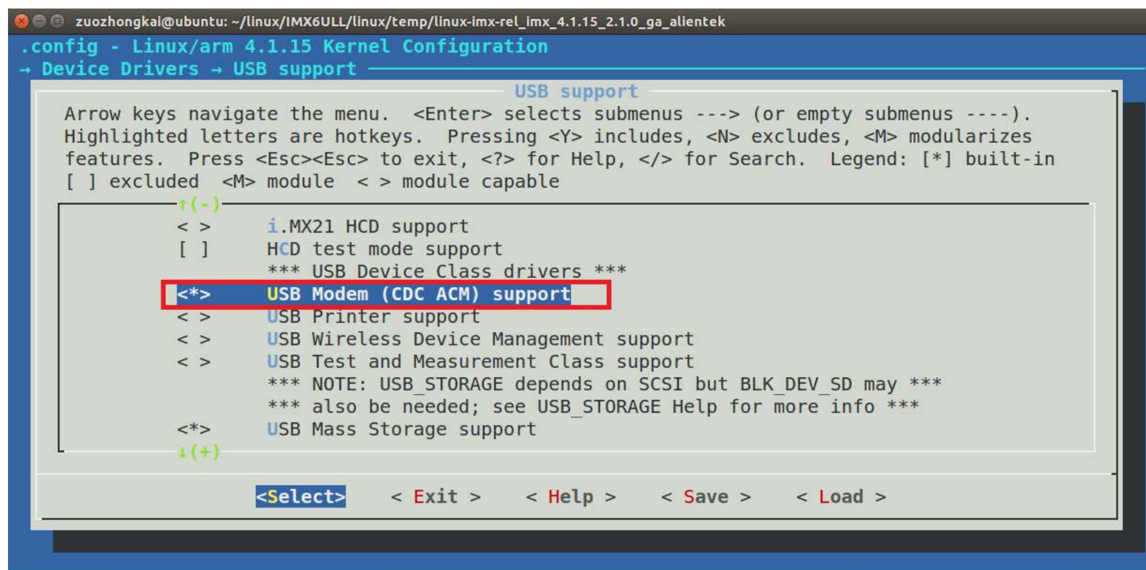


图 71.2.2.5 使能 USB 的 CDC ACM 功能

关于 Linux 内核的配置就到此为止，编译一下 Linux 内核，然后使用新的 zImage 启动开发板。如果 ME3630 已经插上的话，系统启动以后就会输出如图 71.2.2.6 所示的信息：

```
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB0
option 2-1.2:1.1: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB1
option 2-1.2:1.2: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB2
```

图 71.2.2.6 ME3630 虚拟 USB 信息

从图 71.2.2.6 可以看出，ME3630 虚拟出了 3 个 USB 设备，分别为 ttyUSB0~ttyUSB2。对于支持 ECM 接口的 4G 模块来讲，比如 ZM5330/ZM8620/ME3620/ME3630。如果模块工作在 ECM 模式下，可以通过运行“ifconfig -a”命令查看对应的网卡，网卡的名字可能为 usbX/ecmX/ethX 等，X 是具体的数字，如果存在的话就说明 ECM 接口驱动加载成功。输入“ifconfig -a”命令，会发现多了一个名为“usb0”的网卡，图 71.2.2.7 所示：

```
usb0      Link encap:Ethernet  HWaddr 2A:07:47:62:3F:87
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

图 71.2.2.7 ME3630 对应的 usb0 网卡

71.2.3 ME3630 4G 模块 ppp 联网测试

1、使能 Linux 内核 ppp 功能

ME3630 支持通过 ppp 拨号上网，也是支持使用 ECM 接口上网，我们先来学习一下如何通过 ppp 拨号上网。首先我们需要配置 Linux 内核，打开 Linux 内核的 ppp 功能，配置路径如下：

```
-> Device Drivers
    -> [*] Network device support
        -> <*> PPP (point-to-point protocol) support
            -> <*> PPP BSD-Compress compression
```

-> <*> PPP Deflate compression
-> [*] PPP filtering
-> <*> PPP MPPE compression (encryption)
-> [*] PPP multilink support
-> <*> PPP over Ethernet
-> <*> PPP support for async serial ports
-> <*> PPP support for sync tty ports

配置完成以后如图 71.2.3.1 所示：

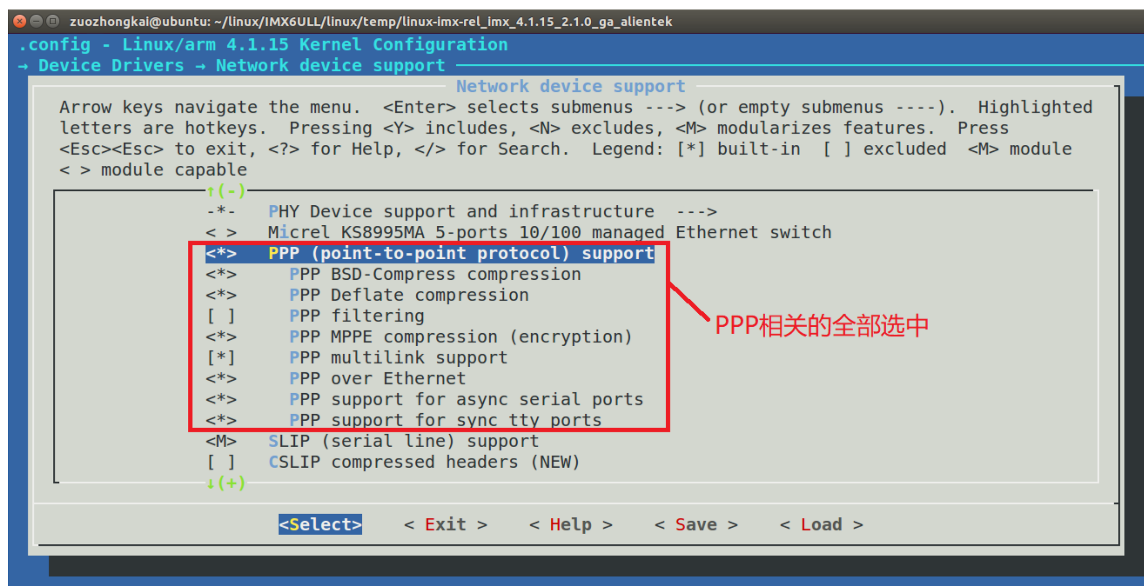


图 71.2.3.1 Linux 内核 PPP 使能

配置完成以后重新编译一下 Linux 内核，得到新的 zImage 镜像文件，然后使用新的 zImage 镜像文件启动开发板。

2、移植 pppd 软件

我们需要通过 pppd 这个软件来实现 ppp 拨号上网，这个软件需要我们移植。

在移植之前先删除掉/usr/sbin/chat 这个软件！

在移植之前先删除掉/usr/sbin/chat 这个软件！

在移植之前先删除掉/usr/sbin/chat 这个软件！

我们使用 Busybox 制作根文件系统的时候会生成/usr/sbin/chat 这个软件，我们一会移植 pppd 的时候也会编译出 chat 软件。因此需要将根文件系统中原来的/usr/sbin/chat 软件删除掉，否则的话我们移植的 chat 软件工作将会出问题！

pppd 源码已经放到了开发板光盘中，路径为：1、例程源码->7、第三方库源码-> ppp-2.4.7.tar.gz，将 ppp-2.4.7.tar.gz 拷贝到 Ubuntu 下并解压，解压以后会生成一个名为 ppp-2.4.7 的文件夹。进入到 ppp-2.4.7 目录中，然后编译 pppd 源码，命令如下：

```
cd ppp-2.4.7/
make CC=arm-linux-gnueabihf-gcc                //编译
```

编译完成以后就会在当前目录下生成 chat/chat、pppd/pppd、pppdump/pppdump 和 pppstats/pppstats 这四个文件，将这个四个文件拷贝到开发板根文件系统中的/sur/bin 目录下，命令如下：

```
sudo cp chat/chat /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
```

```
sudo cp pppd/pppd /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
sudo cp pppdump/pppdump /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
sudo cp pppstats/pppstats /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
```

完成以后输入“pppd -v”查看一下 pppd 的版本号，如果 pppd 版本号显示正常的话就说明 pppd 移植成功，如图 71.2.3.2 所示：

```
/ # pppd -v
pppd: unrecognized option '-v'
pppd version 2.4.7
Usage: pppd [ options ], where options are:
        <device>        Communicate over the named device
        <speed>         Set the baud rate to <speed>
        <loc>:<rem>     Set the local and/or remote interface IP
                        addresses.  Either one may be omitted.
        asyncmap <n>    Set the desired async map to hex <n>
        auth            Require authentication from peer
        connect <p>     Invoke shell command <p> to set up the serial line
        crtscts         Use hardware RTS/CTS flow control
        defaultroute    Add default route through interface
        file <f>        Take options from file <f>
        modem           Use modem control lines
        mru <n>         Set MRU value to <n> for negotiation
See pppd(8) for more options.
```

图 71.2.3.2 pppd 版本号信息

3、ppp 上网测试

在使用 pppd 进行拨号上网之前需要先创建 4 个文件，这个 4 个文件必须放到同一个目录下。在开发板根文件系统下创建/etc/gosuncn 目录，进入到刚刚创建的/etc/gosuncn 目录下，然后新建一个名为“ppp-on”的 shell 脚本文件，在 ppp-on 文件里面输入如下所示内容：

示例代码 71.2.3.1 ppp-on 文件内容

```
1 #!/bin/sh
2 clear
3 OPTION_FILE="gosuncn_options"
4 DIALER_SCRIPT=$(pwd)/gosuncn_ppp_dialer
5 exec pppd file $OPTION_FILE connect "chat -v -f ${DIALER_SCRIPT}"
```

再新建一个名为“gosuncn_option”的文件，在文件里面输入如下所示内容：

示例代码 71.2.3.2 gosuncn_option 文件内容

```
1  /dev/ttyUSB2
2  115200
3  crtscts
4  modem
5  persist
6  lock
7  noauth
8  noipdefault
9  debug
10 nodetach
11 user Anyname
12 password Anypassword
13 ipcp-accept-local
14 ipcp-accept-remote
```

```
15 defaultroute
16 usepeerdns
17 noccp
18 nobsdcomp
19 novj
20 dump
```

第 1 行，如果是联通或移动的卡就是用 ttyUSB2，如果是电信的卡就是用 ttyUSB0。

第 11~12 行，这两行内容和所使用的卡有关，如果是联通或者移动的卡就按照上面的写，如果是电信的卡，要改为如下所示内容：

```
user card
password card
```

再新建一个名为“gosuncn_ppp_dialer”的文件，输入如下所示内容：

示例代码 71.2.3.3 gosuncn_ppp_dialer 文件内容

```
1  ABORT          "NO CARRIER"
2  ABORT          "ERROR"
3  TIMEOUT        120
4  ""             ATE
5  SAY            "ATE"
6  ECHO           ON
7  OK             ATH
8  OK             ATP
9  OK             AT+CGDCONT=1,\"IP\",\"3GNET\"
10 OK             ATD*99#
11 CONNECT
```

第 9 行，后面的 3GNET 是网络的 APN 码，这个要根据自己所使用的手机卡来确定，联通卡的 APN 为 3GNET，移动卡的 APN 为 CMNET。因为我使用的是联通卡进行测试的，所有这里设置 APN 为 3GNET，如果使用的移动卡，那么要将 APN 设置为 CMNET。如果是电信的卡，那么第 9 行要改为：

```
OK             "AT+ZCAPN=card,card"
```

第 10 行，如果是联通或移动的卡，那么第 10 行就不变。如果是电信的卡，那么第 10 行要改为：

```
OK             ATD#777
```

最后新建一个名为“disconnect”的 shell 脚本，输入如下所示内容：

示例代码 71.2.3.4 disconnect 文件内容

```
1 #!/bin/sh
2 killall pppd
```

这四个文件编写完成以后要给予 ppp-on 和 disconnect 这两个文件可执行权限，命令如下：

```
chmod 777 ppp-on disconnect
```

完成以后输入如下命令连接 4G 网络：

```
./ppp-on &
```

在 ME3630 连接 4G 网络的过程中，可能会出现如图 71.2.3.3 所示的错误提示：

```
usepeerdns              # (from gosuncn_options)
noccp           # (from gosuncn_options)
nobsdcomp               # (from gosuncn_options)
Can't create lock file /var/lock/LCK..ttyUSB2: No such file or directory
```

图 71.2.3.3 ppp 拨号上网错误提示

从图 71.2.3.3 可以看出，提示不能创建“Can't create lock file /var/lock/LCK..ttyUSB2”，检查根文件系统是否存在/var/run 和/var/lock 这两个目录，如果没有的话就手动创建这两个文件夹，命令如下：

```
mkdir /var/run              //创建/var/run 文件夹
mkdir /var/lock             //创建/var/lock 文件夹
```

完成以后重新输入“./pppd-on &”命令连接 4G 网络，连接成功以后会输入如图 71.2.3.4 所示信息：

```
Failed to create /etc/ppp/resolv.conf: No such file or directory
not replacing existing default route via 192.168.1.1
local  IP address 10.151.124.203
remote IP address 10.151.124.204
primary   DNS address 116.116.116.116
secondary DNS address 116.116.116.116
```

提示找不到/etc/ppp/resolv.conf这个文件

4G模块IP地址已经获取到了

图 71.2.3.4 4G 网络连接信息

从图 71.2.3.4 可以看出，在 ME3630 4G 模块联网过程中，需要用到/etc/ppp/resolv.conf 文件，但是我们当前根文件系统没有此文件，所以需要手动创建此文件。创建完成以后重启开发板！开发板重启以后进入到/etc/gosuncn 目录，然后输入如下命令完成拨号上网：

```
./ppp-on &
```

ppp 拨号成功以后就会生成一个名为“ppp0”的网卡，如图 71.2.3.5 所示：

```
ppp0      Link encap:Point-to-Point Protocol
          inet addr:10.34.161.30  P-t-P:10.34.161.29  Mask:255.255.255.255
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:3
          RX bytes:68 (68.0 B)  TX bytes:54 (54.0 B)
```

图 71.2.3.5 ppp0 网卡信息

4G 网络测试需要关闭其他网卡，否则的话网络测试可能有问题，但是我们现在是通过网络启动的系统，并且通过 NFS 挂载的根文件系统，因此我们没法关闭其他的网卡，比如 eth0。为了解决这个问题，我们只能将 uboot、Linux kernel、.dtb 设备树和根文件系统都烧写到板子的 EMMC 或 NAND 上，然后直接启动 EMMC 或 NAND 上的系统即可，这样就不需要其他网卡工作了。烧写方法请参考我们的《第二十九章 系统烧写》，这里就不详细的讲解了。系统烧写完成以后设置开发板从 EMMC 或 NAND 启动，因为我使用的是 EMMC 核心板，因此设置从 EMMC 启动，启动以后按照前面的步骤先让 ME3630 4G 模块连接上网络。确保当前开发板只有一个 ME3630 对应的 ppp0 网卡，最后直接 ping 百度官网即可，结果如图 71.2.3.6 所示：

```
/etc/gosuncn # ping www.baidu.com
PING www.baidu.com (163.177.151.110): 56 data bytes
64 bytes from 163.177.151.110: seq=0 ttl=53 time=29.598 ms
64 bytes from 163.177.151.110: seq=1 ttl=53 time=57.429 ms
64 bytes from 163.177.151.110: seq=2 ttl=53 time=48.761 ms
64 bytes from 163.177.151.110: seq=3 ttl=53 time=48.115 ms
```

图 71.2.3.6 ME3630 4G 模块 ping 百度官网

71.2.4 ME3630 4G 模块 ECM 联网测试

对于支持 ECM 接口的模块可以直接通过 ECM 上网，ME3630 模块支持 ECM 接口，重启开发板，输入“ifconfig -a”命令可以看到有一个名为“usb0”的网卡，如图 71.2.4.1 所示：

```
usb0      Link encap:Ethernet  HWaddr FE:D6:B1:38:B5:6A
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

图 71.2.4.1 usb0 网卡

这个 usb0 网卡就是 ECM 接口对应的网卡，我们需要使用 minicom 输入一些 AT 指令，所以要先用 minicom 打开 ttyUSB1，ttyUSB1 就是 ME3630 的 AT 指令串口，波特率设置为 115200。打开以后依次输入如下指令：

①、输入 AT 指令：

```
AT+ZSWITCH=L
```

然后重启开发板。如果模块已经设置为 ECM 模式的话此步骤就不需要了。

②、使用 AT 指令+CGDCONT 来设置数据参数。联通卡的 APN 为 3gnet，电信卡的 APN 为 ctnet，移动卡的 APN 为 cmnet。比如我现在用的联通卡，所以设置 APN 为 3gnet，命令如下：

```
AT+CGDCONT=1,” IP” ,” CMNET”
```

③、发送连接 AT 命令：

```
AT+ZECMCALL=1
```

等待连接成功，连接成功以后会输出如图 71.2.4.2 所示信息：

```
+ZECMCALL: CONNECT
OK
```

图 71.2.4.2 ME3630 ECM 接口网路连接成功

连接成功以后打开 usb0 网卡，命令如下：

```
ifconfig usb0 up            //打开 usb0 网卡
```

usb0 网卡打开以后输入如下命令获取 IP 地址：

```
udhcpc -i usb0
```

IP 地址获取过程如图 71.2.4.3 所示：

```
/ # udhcpc -i usb0
udhcpc: started, v1.29.0
Setting IP address 0.0.0.0 on usb0
udhcpc: sending discover
udhcpc: sending discover
udhcpc: sending select for 10.164.232.36
udhcpc: lease of 10.164.232.36 obtained, lease time 43200
Setting IP address 10.164.232.36 on usb0
Deleting routers
route: SIOCDELRT: No such process
Adding router 10.164.232.37
Recreating /etc/resolv.conf
 Adding DNS server 120.80.80.80
 Adding DNS server 221.5.88.88
```

图 71.2.4.3 usb0 网卡获取 IP 地址过程

从图 71.2.4.3 可以看出，usb0 网卡获取到的 IP 地址为 10.164.232.36，然后 ping 一下百度官网，如果能 ping 通就说明 ME3630 的 ECM 接口联网成功。如果提示“bad address 'www.baidu.com'”，那么请检查一下 DNS 服务器地址设置是否正确，打开/etc/resolv.conf 文件，然后加入“nameserver 114.114.114.114”即可。

至此 ME3630 4G 模块的网络连接就已经全部测试完成，大家既可以在正点原子的 I.MX6U-ALPHA 开发板上使用 4G 上网了。

71.3 EC20 4G 模块实验

71.3.1 EC20 4G 模块简介

关于 EC20 4G 模块的详细资料请找卖家索要！

EC20 有多种不同的配置，比如全网通纯数据版本、语音版、带 GNSS 版等等，建议大家购买的时候至少要选择全网通数据版，因为我们使用 4G 模块主要还是用于数据通信的。移远的 EC20 4G 模块采用 LTE 3GPP Rel.11 技术，支持最大下行速率 150Mbps，最大上行速率 50Mbps。EC20 4G 模块特性如下：

1、一路 USB2.0 高速接口，最高可达 480Mbps。

2、一组模拟语音接口(可选)。

3、1.8V/3.0V SIM 接口。

4、1 个 UART 接口。

5、W_DISABLE#(飞行模式控制)。

6、LED_WWAN#(网络状态指示)。

EC20 也支持 AT 指令，本教程不讲 AT 指令，关于 AT 指令的使用请参考 EC20 的相关文档。在正式使用 EC20 4G 模块之前，请先将其插入到开发板的 MiniPCIE 座上、上紧螺、插入 Nano SIM 卡、接上天线，如图 71.3.1.1 所示：

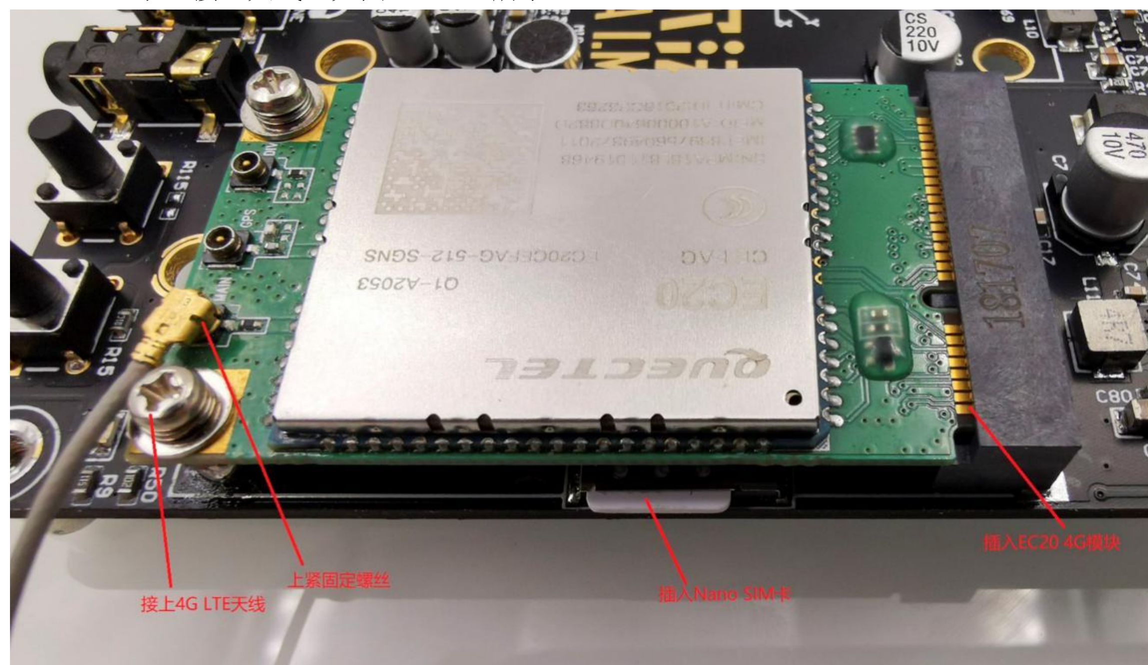


图 71.3.1.1 连接好 4G 模块

一切准备就绪以后就可以开始驱动 EC20 4G 模块了。

71.3.2 EC20 4G 模块驱动修改

1、添加 USB 设备信息

我们需要先在 Linux 内核中添加 EC20 的 USB 设备信息，因为我们前面说了，EC20 4G 模块用的 USB 接口。打开 Linux 源码的 drivers/usb/serial/option.c 文件，首先定义 EC20 的 ID 宏，内容如下：

示例代码 71.3.2.1 EC20 4G 模块 ID

```
1 /* EC20 4G */
2 #define QUECTEL_VENDOR_ID           0X2C7C
3 #define QUECTEL_PRODUCT_EC20        0X0125
```

完成以后如图 71.3.2.1 所示：

```
511
512 /* VIA Telecom */
513 #define VIATELECOM_VENDOR_ID            0x15eb
514 #define VIATELECOM_PRODUCT_CDS7         0x0001
515
516 /* EC20 4G */
517 #define QUECTEL_VENDOR_ID           0X2C7C
518 #define QUECTEL_PRODUCT_EC20        0X0125
```

EC20 4G ID信息

图 71.3.2.1 EC20 ID 添加位置

继续在 drivers/usb/serial/option.c 文件里面找到 option_ids 数组，在此数组里面加入如下内容：

示例代码 71.3.2.2 option_ids 数组添加 EC20 ID 信息

```
{ USB_DEVICE(QUECTEL_VENDOR_ID, QUECTEL_PRODUCT_EC20) }, /* EC20 4G */
```

完成以后 option_ids 数组如图 71.2.2.2 所示：

```
630 static const struct usb_device_id option_ids[] = {
631     { USB_DEVICE(QUECTEL_VENDOR_ID, QUECTEL_PRODUCT_EC20) }, /* EC20 4G */
632     { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_COLT) },
633     { USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA) },
```

图 71.3.2.2 修改后的 option_ids 数组

继续在 drivers/usb/serial/option.c 文件里面找到 option_probe 函数，在此函数里面添加如下内容：

示例代码 71.3.2.3 option_probe 函数添加的代码

```
1  /* EC20  */
2  if (dev_desc->idVendor == cpu_to_le16(0x05c6) &&
3      dev_desc->idProduct == cpu_to_le16(0x9003) &&
4      iface_desc->bInterfaceNumber >= 4)
5      return -ENODEV;
6
7  if (dev_desc->idVendor == cpu_to_le16(0x05c6) &&
8      dev_desc->idProduct == cpu_to_le16(0x9215) &&
9      iface_desc->bInterfaceNumber >= 4)
10     return -ENODEV;
11
```

```
12 if (dev_desc->idVendor == cpu_to_le16(0x2c7c) &&
13     iface_desc->bInterfaceNumber >= 4)
14     return -ENODEV;
```

添加完成以后的 option_probe 函数如图 71.3.2.3 所示：

```
1885    if (dev_desc->idVendor == cpu_to_le16(SAMSUNG_VENDOR_ID) &&
1886        dev_desc->idProduct == cpu_to_le16(SAMSUNG_PRODUCT_GT_B3730) &&
1887        iface_desc->bInterfaceClass != USB_CLASS_CDC_DATA)
1888        return -ENODEV;
1889
1890    /* EC20  */
1891    if (dev_desc->idVendor == cpu_to_le16(0x05c6) &&
1892        dev_desc->idProduct == cpu_to_le16(0x9003) &&
1893        iface_desc->bInterfaceNumber >= 4)
1894        return -ENODEV;
1895
1896    if (dev_desc->idVendor == cpu_to_le16(0x05c6) &&
1897        dev_desc->idProduct == cpu_to_le16(0x9215) &&
1898        iface_desc->bInterfaceNumber >= 4)
1899        return -ENODEV;
1900
1901    if (dev_desc->idVendor == cpu_to_le16(0x2c7c) &&
1902        iface_desc->bInterfaceNumber >= 4)
1903        return -ENODEV;
1904
1905    /* Store the blacklist info so we can use it during attach. */
1906    usb_set_serial_data(serial, (void *)blacklist);
```

添加到option_probe中的代码

图 71.3.2.3 修改后的 option_probe 函数

继续在 drivers/usb/serial/option.c 文件里面找到 option_1port_device 结构体变量，在里面加入休眠后唤醒接口，如图 71.3.2.4 所示：

```
1843     .read_int_callback = option_instat_callback,
1844 #ifdef CONFIG_PM
1845     .suspend           = usb_wwan_suspend,
1846     .resume            = usb_wwan_resume,
1847     .reset_resume      = usb_wwan_resume,
1848 #endif
1849 };
```

添加的休眠唤醒接口

图 71.3.2.4 添加的休眠唤醒接口

打开 drivers/usb/serial/usb_wwan.c 文件，在 usb_wwan_setup_urb 函数中添加零包处理代码，完成后的 usb_wwan_setup_urb 函数如下所示：

示例代码 71.3.2.4 修改后的 usb_wwan_setup_urb 函数

```
1 static struct urb *usb_wwan_setup_urb(struct usb_serial_port *port,
2                   int endpoint,
3                   int dir, void *ctx, char *buf, int len,
4                   void (*callback) (struct urb *))
5 {
6     struct usb_serial *serial = port->serial;
7     struct urb *urb;
```

```
8
9       urb = usb_alloc_urb(0, GFP_KERNEL); /* No ISO */
10      if (!urb)
11          return NULL;
12
13      usb_fill_bulk_urb(urb, serial->dev,
14           usb_sndbulkpipe(serial->dev, endpoint) | dir,
15           buf, len, callback, ctx);
16
17      /* EC20 */
18      if (dir == USB_DIR_OUT) {
19          struct usb_device_descriptor *desc =
                    &serial->dev->descriptor;
20          if (desc->idVendor == cpu_to_le16(0x05c6) &&
21              desc->iProduct == cpu_to_le16(0x9090))
22              urb->transfer_flags |= URB_ZERO_PACKET;
23
24          if (desc->idVendor == cpu_to_le16(0x05c6) &&
25              desc->iProduct == cpu_to_le16(0x9003))
26              urb->transfer_flags |= URB_ZERO_PACKET;
27
28          if (desc->idVendor == cpu_to_le16(0x05c6) &&
29              desc->iProduct == cpu_to_le16(0x9215))
30              urb->transfer_flags |= URB_ZERO_PACKET;
31
32          if (desc->idVendor == cpu_to_le16(0x2c7c))
33              urb->transfer_flags |= URB_ZERO_PACKET;
34      }
35
36      return urb;
37 }
```

第 18~34 行就是要添加到 usb_wwan_setup_urb 函数里面的零包处理代码。至此，Linux 内核需要修改的地方就完了。编译一下 Linux 内核，检查一下代码修改有没有问题，如果修改正确的话就是没有任何问题的。

2、配置 Linux 内核

我们需要配置 Linux 内核，使能 USB NET、GSM、CDMA 驱动等，配置方法和我们在 71.2.2 小节讲解的 ME3630 4G 模块的配置方法一样，大家参考 71.2.2 小节即可。配置完成以后编译一下 Linux 内核，然后使用新的 zImage 启动开发板。如果 EC20 已经插上的话，系统启动以后就会输出如图 71.3.2.5 所示的信息：

```
usb 2-1.2: new high-speed USB device number 3 using ci_hdrc
option 2-1.2:1.0: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB0
option 2-1.2:1.1: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB1
option 2-1.2:1.2: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB2
option 2-1.2:1.3: GSM modem (1-port) converter detected
usb 2-1.2: GSM modem (1-port) converter now attached to ttyUSB3
```

图 71.3.2.5 EC20 虚拟出的 USB 接口

从图 71.3.2.5 可以看出，多了 ttyUSB0~ttyUSB3，这 4 个 tty 接口就是 EC20 虚拟出来的，可以查看一下/dev/ttyUSB0~ttyUSB3 是否存在，结果如图 71.3.2.6 所示：

```
/ # ls /dev/ttyUSB*
/dev/ttyUSB0  /dev/ttyUSB1  /dev/ttyUSB2  /dev/ttyUSB3
/ #
```

图 71.3.2.6 EC20 虚拟出来的 tty 接口

图 71.3.2.6 中的这 4 路 ttyUSB 的含义见表 71.3.2.1：

ttyUSB	描述
ttyUSB0	DM
ttyUSB1	GPS 的 NMEA 信息输出接口
ttyUSB2	AT 指令接口
ttyUSB3	PPP 连接或 AT 指令接口

表 71.3.2.1 四路 ttyUSB 函数

我们用到最多的就是 ttyUSB1 和 ttyUSB2，如果你购买的 EC20 模块带有 GPS 功能，那么就可以通过 ttyUSB1 接口读取 GPS 数据。如果你想使用 EC20 的 AT 指令功能，那么就使用 ttyUSB2 接口即可。

3、添加移远官方的 GobiNet 驱动

移远为 EC20 提供了 GobiNet 驱动，驱动源码我们已经放到了开发板光盘中，路径为：1、例程源码->5、模块驱动源码->3、4G 模块->移远 EC20\EC20_R2.1_Mini_PCIe-C->05 Driver->Linux->GobiNet-> Quectel_WCDMA&LTE_Linux&Android_GobiNet_Driver_V1.3.0.zip。将 Quectel_WCDMA&LTE_Linux&Android_GobiNet_Driver_V1.3.0/src 下的所有.c 和.h 文件都拷贝到 Linux 内核中的/driver/net/usb 目录下，也就是图 71.3.2.7 中所示的文件：

20_R2.1_Mini_PCIe-C › 05 Driver › Linux › GobiNet › Quectel_WCDMA&LTE_Linux&Android_GobiNet_Driver_V1.3.0 › src

名称	修改日期	类型	大小
GobiUSBNet.c	2017-08-15 15:58	sourceinsight.c_file	50 KB
QMI.c	2016-10-25 13:04	sourceinsight.c_file	39 KB
QMI.h	2016-10-25 13:04	H 文件	10 KB
QMIDevice.c	2016-10-26 11:46	sourceinsight.c_file	113 KB
QMIDevice.h	2016-10-25 13:04	H 文件	11 KB
Structs.h	2017-02-24 10:37	H 文件	14 KB

图 71.3.2.7 需要拷贝的文件

拷贝完成以后打开 Linux 内核的 drivers/net/usb/Makefile 文件，在此文件末尾加入如下内容：

示例代码 71.3.2.5 修改后的 drivers/net/usb/Makefile 文件

```
1 obj-$(CONFIG_USB_GOBI_NET) += GobiNet.o
2 GobiNet-objs := GobiUSBNet.o QMIDevice.o QMI.o
```

最后在 drivers/net/usb/Kconfig 文件中加入下所示内容：

示例代码 71.3.2.6 drivers/net/usb/Kconfig 要添加的内容

```
1 config USB_GOBI_NET
2   tristate"Gobi USB Net driver for Quectel module"
3   help
4   Support Quectelmodule.
5 
6   A modemmanager with support for GobiNet is recommended.
7   Tocompile this driver as a module, choose M here: the module will be calledGobiNet.1
```

上述内容在 drivers/net/usb/Kconfig 文件中的位置如图 71.3.2.8 所示：

```
575
576 config USB_GOBI_NET
577     tristate"Gobi USB Net driver for Quectel module"
578     help
579     Support Quectelmodule.
580
581     A modemmanager with support for GobiNet is recommended.
582     Tocompile this driver as a module, choose M here: the module will be calledGobiNet.
583
584 endif # USB_NET_DRIVERS
585
```

添加的内容

图 71.3.2.8 内容要添加的位置

完成以后打开 Linux 内核配置界面，使能前面添加的 Gobi 驱动，配置路径如下：

```
-> Device Drivers
    -> [*] Network device support
        -> -*- USB Network Adapters
            -> <*> Gobi USB Net driver for Quectel module
```

配置如图 71.3.2.9 所示：

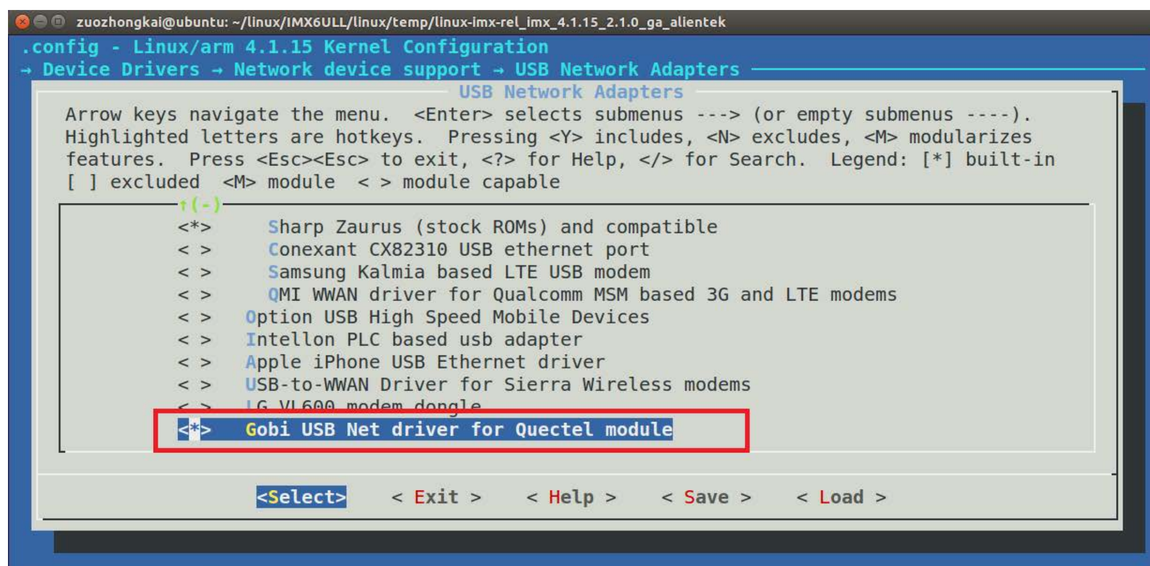


图 71.3.2.9 使能 Gobi 驱动

配置完成以后就重新编译一下 Linux 内核，然后使用新的 zImage 启动开发板。启动以后检查一下“/dev/qcqmi2”这个文件是否存在，如果存在的话就说明 Gobi 驱动工作成功。至此，EC20 的驱动就已经修改完成了，接下来就是使用 EC20 来实现联网。

71.3.3 quectel-CM 移植

要使用 EC20，我们需要用到移远提供的 quectel-CM 这个软件，这个软件是移远提供的网络管理工具，软件源码已经放到了开发板光盘中，路径为：1、例程源码->5、模块驱动源码->3、4G 模块->移远 EC20->EC20_R2.1_Mini_PCIe-C->05 Driver->Linux->GobiNet->WCDMA&LTE_QConnectManager_Linux&Android_V1.1.34.zip。将 WCDMA&LTE_QConnectManager_Linux&Android_V1.1.34.zip 这个压缩包进行解压，得到 quectel-CM 这个文件夹，然后将 quectel-CM 文件夹拷贝到 Ubuntu 中。拷贝完成以后进入到 Ubuntu 中的 quectel-CM 文件夹，使用如下命令进行交叉编译：

```
make CROSS_COMPILE=arm-linux-gnueabihf-
```

编译完成以后得到一个名为“quectel-CM”软件，如图 71.3.3.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/quectel-CM$ ls
default.script  main.c     MPQMI.h   QMIThread.c  quectel-CM  util.h
dhcpclient.c    Makefile   MPQMUX.c  QMIThread.h  udhcpc.c
GobiNetCM.c     MPQCTL.h   MPQMUX.h  QmiWwanCM.c  util.c
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/quectel-CM$
```

图 71.3.3.1 编译出来的 quectel-CM 软件

将图 71.3.3.1 中编译出来的 quectel-CM 软件拷贝到开发板根文件系统的/usr/bin 目录下，命令如下：

```
sudo cp quectel-CM /home/zuozhongkai/linux/nfs/rootfs/usr/bin/ -f
```

拷贝完成以后就可以使用 quectel-CM 软件来实现 EC20 联网测试了。

71.3.4 EC20 上网测试

在开发板上输入如下命令完成 EC20 的 4G 网络连接：

```
quectel-CM -s cenet &
```

注意，quectel-CM 软件会使用到 udhcpc 来获取 IP 地址，所以一定要确保当前根文件系统下存在 udhcpc。当 4G 网络连接成功以后就会获取到 IP 地址，如图 71.3.4.1 所示：

```
[01-02_00:10:46:318] busybox udhcpc -f -n -q -t 5 -i eth2
udhcpc: started, v1.29.0
[01-02_00:10:46:402] Setting IP address 0.0.0.0 on eth2
udhcpc: sending discover
udhcpc: sending select for 10.26.28.49
udhcpc: lease of 10.26.28.49 obtained, lease time 7200
[01-02_00:10:46:603] Setting IP address 10.26.28.49 on eth2
[01-02_00:10:46:633] Deleting routers
route: SIOCDELRT: No such process
[01-02_00:10:46:672] Adding router 10.26.28.50
[01-02_00:10:46:696] Recreating /etc/resolv.conf
[01-02_00:10:46:724]  Adding DNS server 116.116.116.116
[01-02_00:10:46:731]  Adding DNS server 221.5.88.88
```

图 71.3.4.1 EC20 4G 网络连接成功并获取到 IP 地址

从图 71.3.4.1 可以看出，EC20 对应的网卡名字为“eth2”，输入“ifconfig eth2”即可查看 eth2 网卡的详细信息，如图 71.3.4.2 所示：

```
/ # ifconfig eth2
eth2      Link encap:Ethernet  HWaddr DA:0D:91:63:8E:49
          inet addr:10.26.28.49  Bcast:10.26.28.51  Mask:255.255.255.252
          inet6 addr: fe80::d80d:91ff:fe63:8e49/64 Scope:Link
          UP BROADCAST RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:612 (612.0 B)  TX bytes:824 (824.0 B)

/ #
```

图 71.3.4.2 eth2 网卡详细信息

4G 网络测试需要关闭其他网卡，否则的话网络测试可能有问题，但是我们现在是通过网络启动的系统，并且通过 NFS 挂载的根文件系统，因此我们没法关闭其他的网卡，比如 eth0。为了解决这个问题，我们只能将 uboot、Linux kernel、.dtb 设备树和根文件系统都烧写到板子的 EMMC 或 NAND 上，然后直接启动 EMMC 或 NAND 上的系统即可，这样就不需要其他网卡工作了。烧写方法请参考我们的《第二十九章 系统烧写》，这里就不详细的讲解了。系统烧写完成以后设置开发板从 EMMC 或 NAND 启动，因为我使用的是 EMMC 核心板，因此设置从 EMMC 启动，启动以后按照前面的步骤先让 EC20 连接上网络。确保当前开发板只有一个 EC20 对应的 eth2 网卡，然后直接直接 ping 百度官网即可，结果如图 71.3.4.3 所示：

```
/ # ping www.baidu.com
PING www.baidu.com (163.177.151.110): 56 data bytes
64 bytes from 163.177.151.110: seq=0 ttl=54 time=18.342 ms
64 bytes from 163.177.151.110: seq=1 ttl=54 time=31.896 ms
64 bytes from 163.177.151.110: seq=2 ttl=54 time=31.401 ms
64 bytes from 163.177.151.110: seq=3 ttl=54 time=30.934 ms
64 bytes from 163.177.151.110: seq=4 ttl=54 time=30.339 ms
```

图 71.3.4.3 ping 百度官网成功

如果 ping 百度官网没有问题，那么就表示 EC20 4G 模块在我们的 I.MX6U-ALPHA 开发板上工作正常。

附录 A

第 A1 章 Buildroot 根文件系统构建

前面我们学习了如何使用 busybox 来构建根文件系统，但是 busybox 构建的根文件系统不其全，很多东西需要我们自行添加，比如 lib 库文件。在我们后面的驱动开发中很多第三方软件也需要我们自己去移植，这些第三方软件有很多又依赖其他的库文件，导致移植过程非常的繁琐。本章我们来学习一下另外一种实用的根文件系统构建方法，那就是使用 buildroot 来构建根文件系统。

A1.1 何为 buildroot?

A1.1.1 buildroot 简介

在《第三篇 系统移植篇》我们最后讲解了如何使用 busybox 构建文件系统，busybox 仅仅只是帮我们构建好了一些常用的命令和文件，像 lib 库、/etc 目录下的一些文件都需要我们自己手动创建，而且 busybox 构建的根文件系统默认没有用户名和密码设置。在后续的实验中，我们还要自己去移植一些第三方软件和库，比如 alsa、iperf、mplayer 等等。那么有没有一种傻瓜式的方法或软件，它不仅包含了 busybox 的功能，而且里面还集成了各种软件，需要什么软件就选择什么软件，不需要我们去移植。答案肯定是有的，buildroot 就是这样一种工具，buildroot 比 busybox 更上一层楼，buildroot 不仅集成了 busybox，而且还集成了各种常见的第三方库和软件，需要什么就选择什么，就跟我们去吃自助餐一样，想吃什么就拿什么。buildroot 极大的方便了我们嵌入式 Linux 开发人员构建实用的根文件系统。

从 busybox 开始一步一步的构建根文件系统适合学习、了解根文件系统的组成，但是不适合做产品(主要是自己构建的话会有很多不完善、没有注意到的细节)。buildroot 会帮我们处理好各种细节地方，是我们的根文件系统更加的合理、有效。因此大家在做产品的时候推荐大家使用 buildroot 来构建自己的根文件系统，当然了，类似 buildroot 的软件还有很多，比如后面要讲的 yocto。

buildroot 和 uboot、Linux Kernel 很类似，我们需要到其官网上下载源码，然后对其进行配置，比如设置交叉编译器、设置目标 CPU 参数等，最主要的就是选择所需要的第三方库或软件。一切配置好以后就可以进行编译，编译完成了以后就会在一个文件夹里面存放好编译结果，也就是根文件系统。

A1.1.2 buildroot 下载

buildroot 源码肯定是要从 buildroot 官网下载，官网地址为 https://buildroot.org/，打开以后的官网界面如图 A1.1.2.1 所示：

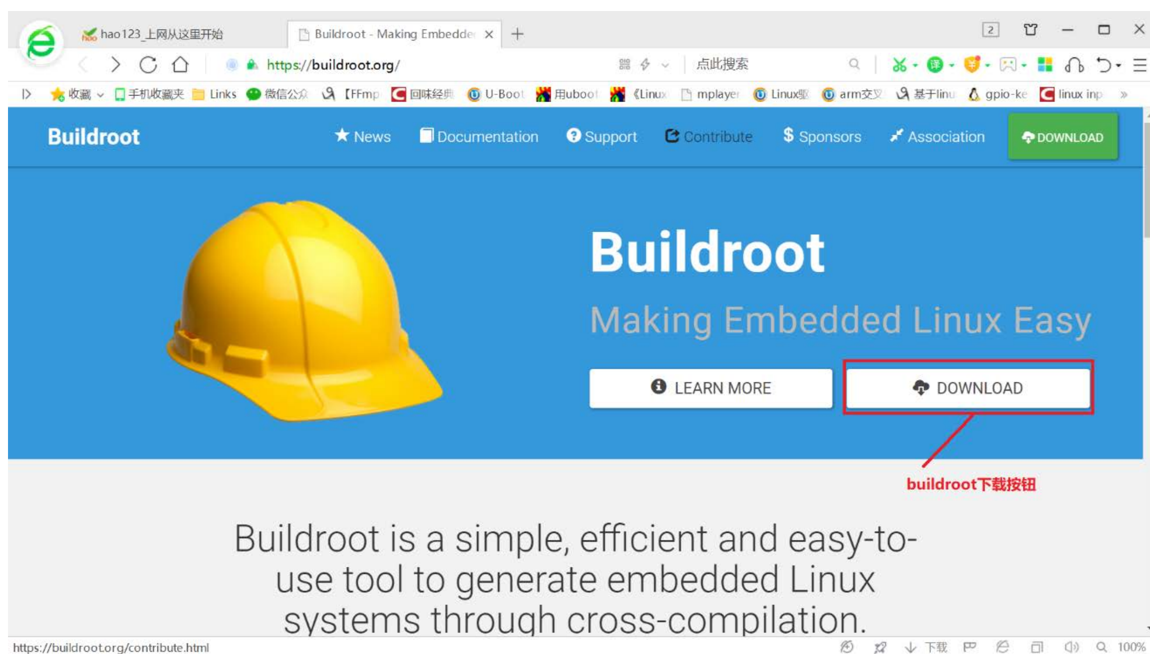


图 A1.1.2.1 buildroot 官网界面

点击图 A1.1.2.1 中的“DOWNLOAD”按钮即可打开 buildroot 的下载界面，如图 A1.1.2.2 所示：

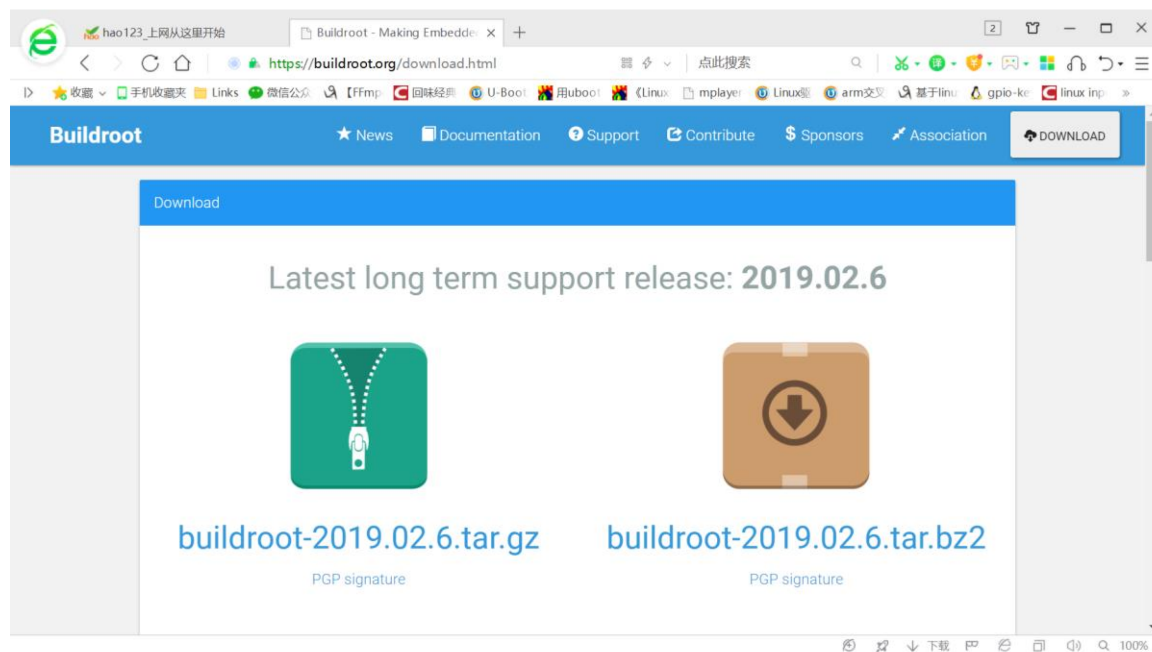


图 A1.1.2.2 buildroot 下载界面

可以看出，在写本教程的时候最新的 LTS(长期支持版)版 buildroot 为 2019.02.6，分为.gz 和.bz2 两种压缩格式，这里我就使用右侧的.bz2 压缩格式的源码，选中以后下载即可。我们已经将其放到了开发板光盘中，路径为：1、例程源码-》8、buildroot 源码-》buildroot-2019.02.6.tar.bz2，一切准备好以后就可以使用 buildroot 构建根文件系统了。

A1.2 buildroot 构建根文件系统

A1.2.1 配置 buildroot

将 buildroot 源码 buildroot-2019.02.6.tar.bz2 拷贝到 ubuntu 中，也就是我们前面创建的 tool 目录下。拷贝完成以后对其进行解压，命令如下：

```
tar -vxjf buildroot-2019.02.6.tar.bz2
```

解压完成以后就会得到一个名为“buildroot-2019.02.6”的目录，此目录就是我们解压得到的 buildroot 源码，进入到此目录中，此目录下的文件如图 A1.2.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6$ ls
arch   CHANGES          configs     docs   Makefile         README   toolchain
board  Config.in        COPYING     fs     Makefile.legacy  support  utils
boot   Config.in.legacy DEVELOPERS  linux  package          system
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6$
```

图 A1.2.1.1 buildroot 源码文件

buildroot 和 uboot、Linux kernel 一样也支持图形化配置，输入如下命令即可打开图形化配置界：

```
make menuconfig
```

打开以后的图形化配置界面如图 A1.2.1.2 所示：

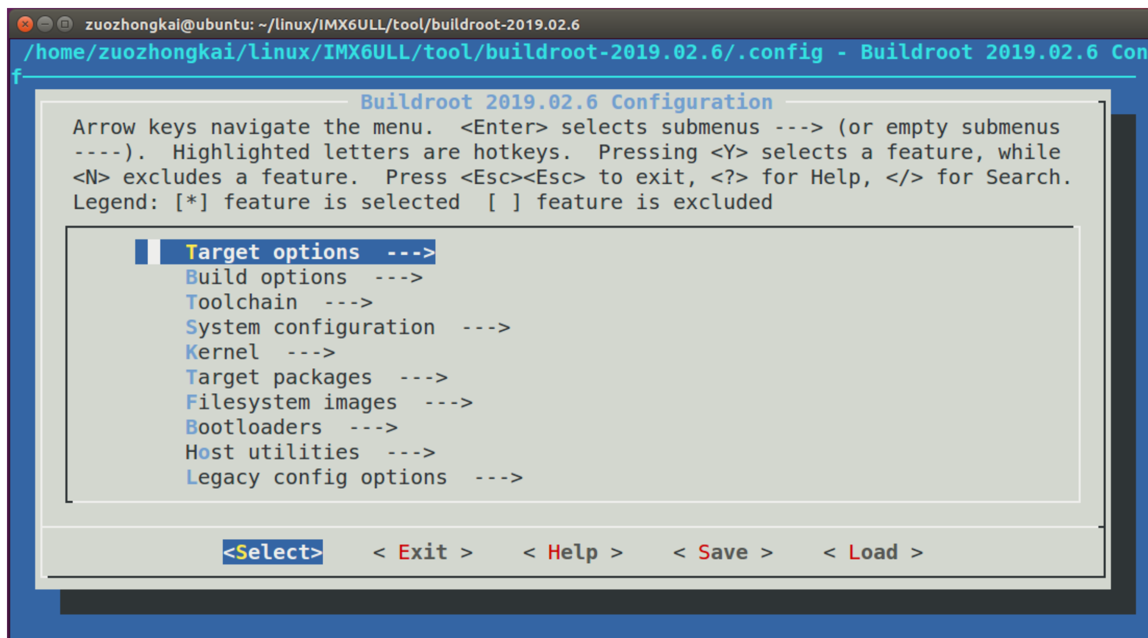


图 A1.2.1.2 buildroot 图形化配置界面

接下来我们就依次配置 buildroot，配置完成以后就可以进行编译了。

1、配置 Target options

首先配置 Target options 选项，需要配置的项目和其对应的内容如下(“=”号后面是配置项要选择的内容！)：

```
Target options
    -> Target Architecture                 = ARM (little endian)
    -> Target Binary Format                = ELF
    -> Target Architecture Variant         = cortex-A7
    -> Target ABI                          = EABIhf
    -> Floating point strategy             = NEON/VFPv4
    -> ARM instruction set                 = ARM
```

配置完成以后如图 A1.2.1.3 所示：

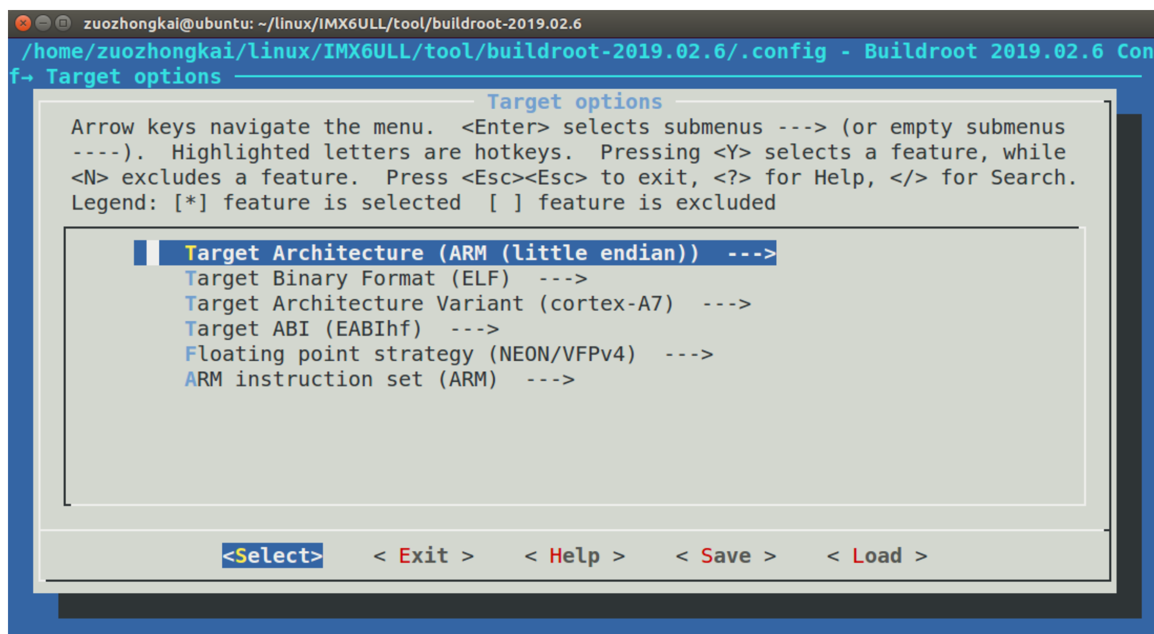


图 A1.2.1.3 配置好的 Target options 选项

2、配置 Toolchain

此配置项用于配置交叉编译工具链，也就是交叉编译器，这里设置为我们自己所使用的交叉编译器即可。buildroot 其实是可以自动下载交叉编译器的，但是都是从国外服务器下载的，鉴于国内的网络环境，强烈推荐大家设置成自己所使用的交叉编译器。需要配置的项目和其对应的内容如下：

```
Toolchain
    -> Toolchain type    = External toolchain
    -> Toolchain         = Custom toolchain          //用户自己的交叉编译器
    -> Toolchain origin = Pre-installed toolchain    //预装的编译器
    -> Toolchain path    =/usr/local/arm/gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf
    -> Toolchain prefix       = $(ARCH)-linux-gnueabihf  //前缀
    -> External toolchain gcc version              = 4.9.x
    -> External toolchain kernel headers series    = 4.1.x
    -> External toolchain C library                = glibc/eglibc
    -> [*] Toolchain has SSP support? (NEW)     //选中
    -> [*] Toolchain has RPC support? (NEW)     //选中
    -> [*] Toolchain has C++ support?           //选中
    -> [*] Enable MMU support (NEW)             //选中
```

Toolchain 下几个比较重要的选项需要说明一下，如下所示：

Toolchain：设置为 Custom toolchain，表示使用用户自己的交叉编译器。

Toolchain origin：设置为 Pre-installed toolchain，表示使用预装的交叉编译器。

Toolchain path：设置自己安装的交叉编译器绝对路径！buildroot 要用到。

Toolchain prefix：设置交叉编译器前缀，要根据自己实际所使用的交叉编译器来设置，比如我们使用的是 arm-linux-gnueabihf-gcc，因此前缀就是$(ARCH)-linux-gnueabihf，其中 ARCH 我们前面已经设置为了 arm。

3、配置 System configuration

此选项用于设置一些系统配置，比如开发板名字、欢迎语、用户名、密码等。需要配置的项目和其对应的内容如下：

```
System configuration
    -> System hostname          = alpha_imx6ull            //平台名字，自行设置
    -> System banner            = Welcome to alpha i.mx6ull //欢迎语
    -> Init system              = BusyBox                  //使用 busybox
    -> /dev management          = Dynamic using devtmpfs + mdev   //使用 mdev
    -> [*] Enable root login with password (NEW)           //使能登录密码
        -> Root password        = 123456                   //登录密码为 123456
```

在 System configuration 选项中可以配置平台名字，登录密码等信息。

5、配置 Filesystem images

此选项配置我们最终制作的根文件系统为什么格式的，配置如下：

```
-> Filesystem images
    -> [*] ext2/3/4 root filesystem       //如果是 EMMC 或 SD 卡的话就用 ext3/ext4
        -> ext2/3/4 variant    = ext4     //选择 ext4 格式
    -> [*] ubi image containing an ubifs root filesystem   //如果使用 NAND 的话就用 ubifs
```

对于 I.MX6U 来说此选项不用配置，因为我们是通过 Mfgtool 工具将根文件系统烧写到开发板上的 EMMC/SD 卡中，烧写的时候需要自己对根文件系统进行打包。

6、配置 Target packages

此选项用于配置要选择的第三方库或软件、比如 alsa-utils、ffmpeg、iperf 等工具，但是现在我们先不选择第三方库，防止编译不下去！先编译一下最基本的根文件系统，如果没有问题的话在重新配置选择第三方库和软件。否则一口吃太多会容易撑着的，编译出问题的时候都不知道怎么找问题。

A1.2.2 编译 buildroot

配置完成以后就可以编译 buildroot 了，编译完成以后 buildroot 就会生成编译出来的根文件系统压缩包，我们可以直接使用。输入如下命令开始编译：

```
sudo make    //注意，一定要加 sudo，而且不能通过-jx 来指定多核编译！！！
```

buildroot 编译的时候会先从网上下载所需的软件源码，有些软件源码可能下载不下来，这个时候就需要我们自行处理，这个后面会详细的讲解。

buildroot 编译过程会很耗时，可能需要几个小时，请耐心等待！

buildroot 编译过程会很耗时，可能需要几个小时，请耐心等待！

buildroot 编译过程会很耗时，可能需要几个小时，请耐心等待！

buildroot 因为要从网上下载源码，因此可能存在有些源码无法下载或下载很慢的情况，比如如图 A1.2.2.1 所示：

```
>>> host-cmake 3.8.2 Downloading
--2019-10-19 00:11:59-- https://cmake.org/files/v3.8/cmake-3.8.2.tar.gz
正在解析主机 cmake.org (cmake.org)... 66.194.253.19
正在连接 cmake.org (cmake.org)|66.194.253.19|:443... 已连接。
已发出 HTTP 请求，正在等待回应... 200 OK
长度： 7504706 (7.2M) [application/x-gzip]
正在保存至："/home/zuozhongkai/linux/IMX6ULL/tool/buildroot-2019.02.6/output/build/.cmake-3.8.2.tar.gz.pUr4jV/output"

/home/zuozhongkai/linux/IMX6ULL/tool/buil    4%[==>
] 328.00K  1.60KB/s    eta 71m 47s
```

从此网站下载cmake-3.8.2.tar.gz源码

需要71分钟

图 A1.2.2.1 下载 cmake-3.8.2.tar.gz

可以看出图 A1.2.2.1 中正在下载 cmake-3.8.2.tar.gz 这个压缩包，大小是 7.2MB，当前下载网速是 1.6KB/S，需要用时 71 分钟，显然这是无法忍受的！我们可以自行到 https://cmake.org/files/v3.8/cmake-3.8.2.tar.gz 这个网站上去将 cmake-3.8.2.tar.gz 这个源码下载下来，然后拷贝到 Ubuntu 中 buildroot 源码目录下的 dl 文件夹中，dl 文件夹专用用于存放下载下来的源码。

等待编译完成，编译完成以后就会在 buildroot-2019.02.6/output/images 下生成根文件系统，如图 A1.2.2.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/output/images$ ls
rootfs.ext2  rootfs.ext4  rootfs.tar  rootfs.ubi  rootfs.ubifs
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/output/images$
```

图 A1.2.2.2 编译生成的根文件系统

从图 A1.2.2.2 可以看出，编译出来了多种格式的 rootfs，比如 ext2、ext4、ubi 等。其中 rootfs.tar 就是打包好的根文件系统，我们就使用 rootfs.tar 进行测试。在 nfs 目录下新建一个名为 buildrootfs 的文件夹，然后将图 A1.2.2.2 中的 rootfs.tar 拷贝到 buildrootfs 目录下并解压，命令如下：

```
cd /home/zuozhongkai/linux/nfs          //进入到 nfs 目录下
mkdir buildrootfs                       //创建 buildrootfs 目录
cd buildrootfs                          //进入到 buildrootfs 目录
cp ../../IMX6ULL/tool/buildroot-2019.02.6/output/images/rootfs.tar ./      //拷贝到 rootfs.tar
tar
tar -vxf rootfs.tar                     //解压缩 rootfs.tar
rm rootfs.tar                           //删除 rootfs.tar
```

解压缩完成以后的 buildrootfs 目录如图 A1.2.2.3 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$ ls
bin  etc  lib32    media  opt   root  sbin  tmp  var
dev  lib  linuxrc  mnt    proc  run   sys   usr
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$
```

图 A1.2.2.3 使用 buildroot 编译出来的根文件系统

图 A1.2.2.3 就是使用 buildroot 编译出来的根文件系统，我们可以通过 nfs 挂载到开发板上，然后对其进行测试。

A2.2.3 buildroot 根文件系统测试

buildroot 制作出来的根文件系统已经准备好了，接下来就是对其进行测试。测试方法也是通过 nfs 挂载的方式，启动 uboot，修改 bootargs 环境变量，设置 nfsroot 目录为 Ubuntu 中的 buildrootfs 目录，命令如下：

```
setenv bootargs’console=tty1 console=ttymxc0,115200 root=/dev/nfs nfsroot=192.168.1.253:/home/zuozhongkai/linux/nfs/buildrootfs rw ip=192.168.1.251:192.168.1.253:192.168.1.1:255.255.255.0::eth0:off’
```

设置好以后启动系统，进入根文件系统以后如图 A2.2.3.1 所示：

```
devpts: called with bogus options
Starting syslogd: OK
Starting klogd: OK
Starting mdev...
modprobe: can't change directory to '/lib/modules': No such file or directory
Initializing random number generator... random: dd urandom read with 68 bits of entropy available
done.
Starting network: ip: RTNETLINK answers: File exists
FAIL

Welcome to alpha i.mx6ull
alpha_imx6ull login:
Welcome to alpha i.mx6ull
alpha_imx6ull login:
Welcome to alpha i.mx6ull
alpha_imx6ull login:
```

提示/lib/modules目录不存在　欢迎语　用户名:root

图 A2.2.3.1 buildroot 根文件系统

在图 A2.2.3.1 中，首先要进入到/lib/modules 目录，但是默认没有，因此需要我们自行创建此目录。buildroot 构建的根文件系统启动以后会输出我们前面设置的欢迎语"Welcome to alpha i.mx6ull"。然后需要输入用户名和密码，用户名是"root"，密码就是我们前面配置 buildroot 的时候设置的"123456"。输入用户名和密码以后就可以进入系统中，如图 A2.2.3.2 所示：

```
Welcome to alpha i.mx6ull
alpha_imx6ull login: root
Password:
# cd /
# ls
bin      lib      media    proc     sbin     usr
dev      lib32    mnt      root     sys      var
etc      linuxrc  opt      run      tmp
#
```

图 A2.2.3.1 buildroot 根文件系统文件

可以看出的 buildroot 构建的根文件系统运行基本没有问题，但是这个根文件系统是最简单的，我们并没有在 buildroot 里面配置任何第三方的库和软件，接下来我们就配置 buildroot，使能一些常见的第三方软件。

A1.3 buildroot 第三方软件和库的配置

我在前面学习的时候需要自行移植一些第三方的库和软件，比如 alsa-lib、alsa-utils 等等，现在我们不需要自行移植这些第三方软件和库了，可以直接在 buildroot 里面配置使能。比如我们现在配置使能 alsa-lib、alsa-utils 这两个软件和库。

1、使能 alsa-lib

输入 make menuconfig，打开 buildroot 配置界面，配置路径如下：

```
Target packages
    -> Libraries
        -> Audio/Sound
            -> -*- alsa-lib  --->   此配置项下的文件全部选中
```

如图 A1.3.1 所示：

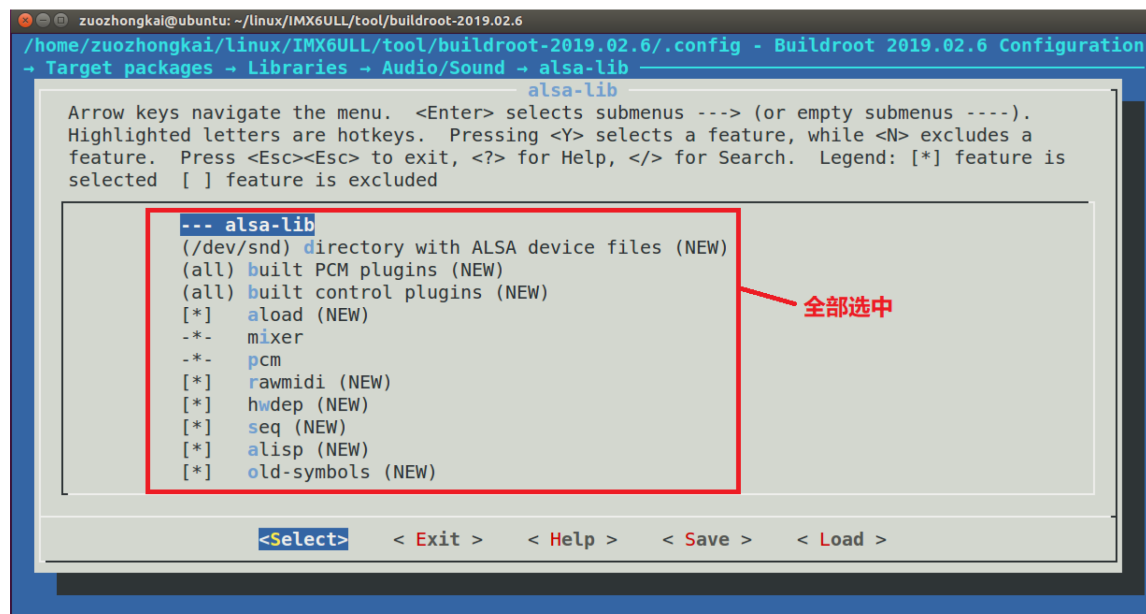


图 A1.3.1 alsa-lib 库文件

2、使能 alsa-utils

接下来使能 alsa-utils，路径如下：

```
Target packages
    -> Audio and video applications
        -> alsa-utils                    此目录下的软件全部选中
```

配置好以后如图 A1.3.2 所示：

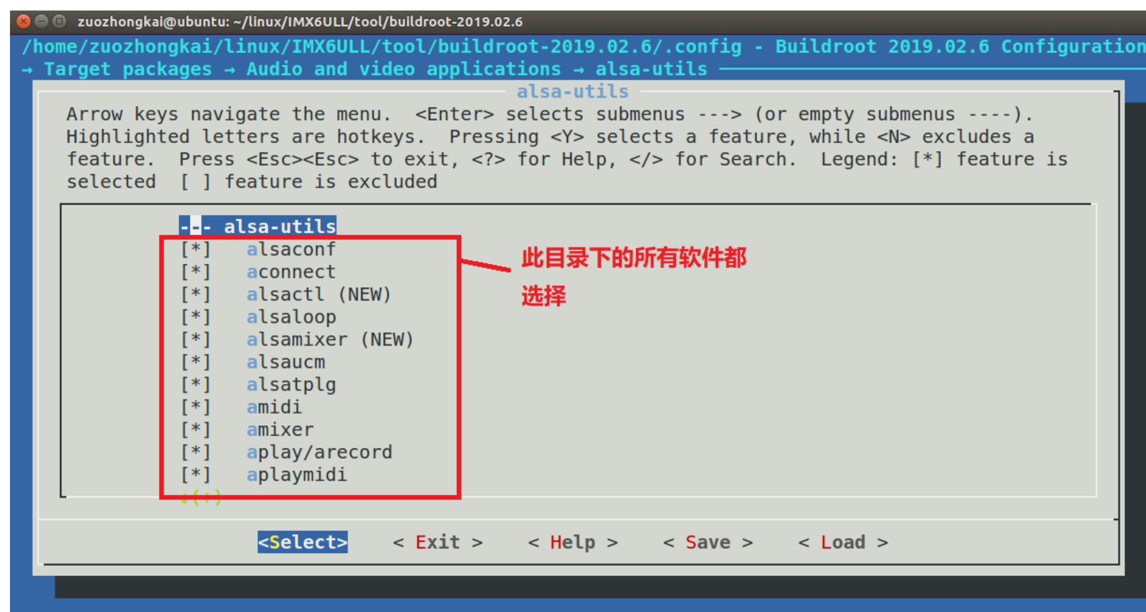


图 A1.3.2 alsa-utils 软件配置

等待编译完成就可以使用新的根文件系统进行测试了，将 buildroot/images/rootfs.tar 拷贝到 nfs 目录下的 buildroot 目录中，然后重新解压。注意，以前自己添加的文件并不会被删除掉的，解压命令如下：

```
tar -vxf rootfs.tar
```

解压完成以后就可以使用 alsa-utils 相关的软件了，比如 alsamixer，想想我们曾经在第六十五章中自行移植 alsa-utils 的时候，那个过程叫一个复杂。通过 buildroot 的话直接一个配置就可以搞定全部，方便快捷，大家可以自行尝试去配置使能一些其他的第三方库和软件。

A1.4 buildroot 下的 busybox 配置

A1.4.1 busybox 配置

buildroot 在构建根文件系统的时候也是要用到 busybox 的，既然用到了 busybox 那么就涉及到 busybox 的配置。buildroot 会自动下载 busybox 压缩包，buildroot 下载的源码压缩包都存放在/dl 目录下，在 dl 目录下就有一个叫做“busybox”的文件夹，此目录下保存着 busybox 压缩包，如图 A1.4.1.1 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/dl/busybox$ ls
busybox-1.29.3.tar.bz2
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/dl/busybox$
```

图 A1.4.1.1 busybox 源码

可以看出，buildroot 下载的 busybox 版本为 1.29.3。要想编译 busybox，必须对图 A1.4.1.1 中的压缩包进行解压缩，buildroot 将所有解压缩后的软件保存在/output/build 软件中，我们可以在找到/output/build/busybox-1.29.3 这个文件夹，此文件夹就是解压后的 busybox 源码，文件内容如图 A1.4.1.2 所示：

```
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/output/build/busybox-1.29.3$ ls
applets               debianutils  mailutils              README
applets_sh            docs         Makefile               runit
arch                  e2fsprogs    Makefile.custom        scripts
archival              editors      Makefile.flags         selinux
AUTHORS               examples     Makefile.help          shell
busybox               findutils    make_single_applets.sh size_single_applets.sh
busybox.links         include      miscutils              sysklogd
busybox_unstripped    init         modutils               testsuite
busybox_unstripped.map INSTALL     networking             TODO
busybox_unstripped.out klibc-utils NOFORK_NOEXEC.lst      TODO_unicode
Config.in             libbb        NOFORK_NOEXEC.sh       util-linux
configs               libpwdgrp    printutils
console-tools         LICENSE      procps
coreutils             loginutils   qemu_multiarch_testing
zuozhongkai@ubuntu:~/linux/IMX6ULL/tool/buildroot-2019.02.6/output/build/busybox-1.29.3$
```

图 A1.4.1.2 busybox 源码

如果大家想要修改 busybox 源码的话就直接在图 A1.4.1.2 中找到相应的文件，然后修改即可。我们现在是要配置 buildroot 下的 busybox，因此肯定要打开 busybox 的配置界面，在 buildroot 下打开 busybox 的配置界面输入如下命令：

```
sudo make busybox-menuconfig
```

输入以后就会打开 buildroot 下的 busybox 配置界面，如图 A1.4.1.3 所示：

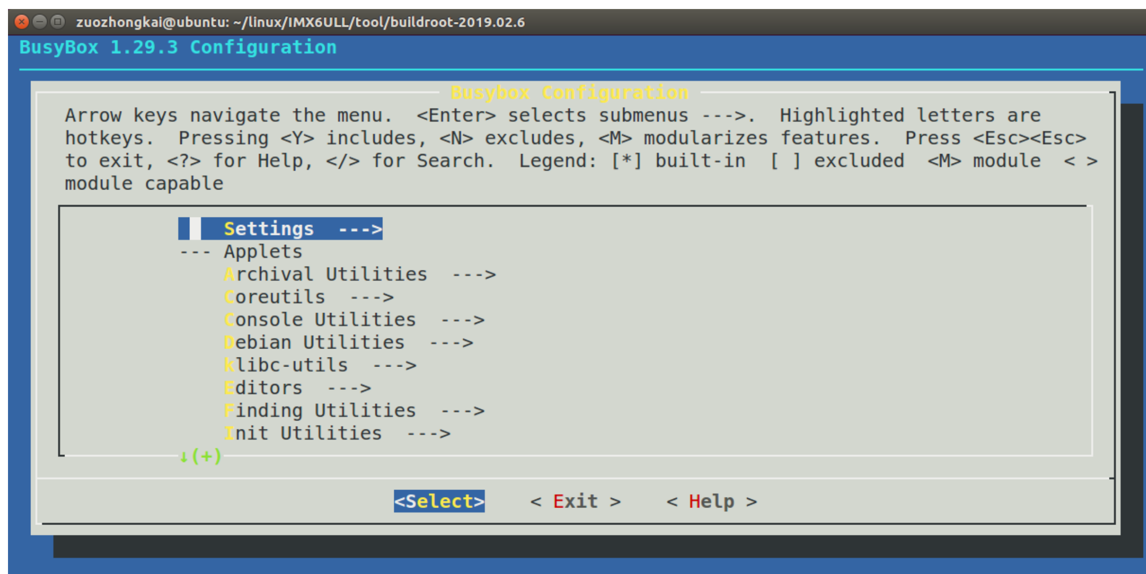


图 A1.4.1.3 buildroot 下的 busybox 配置界面

图 A1.4.1.3 就是我们最熟悉的 busybox 配置界面了，大家想要配置什么就直接操作就行了，或者参考第三十八章即可。

A1.4.2 busybox 中文字符的支持

在第三十八章我们数过了，busybox 对中文字符显示做了限制，因此必须要修改相关的文件，具体修改过程参考 38.2.2 小节即可，这里就不再赘述了。

A1.4.3 编译 busybox

配置好以后就可以重新编译 buildroot 下的 busybox，进入到 buildroot 源码目录下，输入如下命令查看当前 buildroot 所有配置了的目标软件包，也就是 packages：

```
sudo make show-targets
```

结果如图 A1.4.3.1 所示：

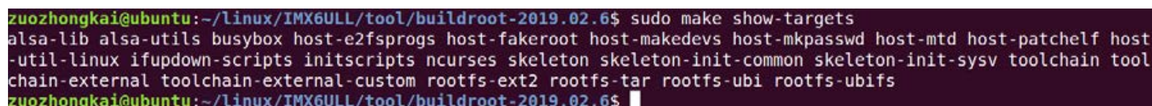


图 A1.4.3.1 当前 buildroot 下的所有 packages

图 A1.4.3.1 中列出了当前 buildroot 中所有使能了的 packages 包，其中就包括 busybox，如果我们想单独编译并安装 busybox 的话执行下面命令即可：

```
sudo make busybox
```

上述命令就会重新编译 busybox。编译完成以后重现编译 buildroot，主要是对其进行打包，输入如下命令：

```
sudo make
```

重新编译完成以后查看一下 output/images 目录下 rootfs.tar 的创建时间是否为刚刚编译的，如果不是的话就删除掉 rootfs.tar，然后重新执行“sudo make”重新编译一下即可。最后我们使用新的 rootfs.tar 启动 Linux 系统。

A1.5 根文件系统测试

buildroot 的根文件系统制作好以后就是测试工作了，测试内容和第三十八章中一样，这里就不赘述了。这里我们重点说一下另外一个问题，我们使用构建的根文件系统启动以后会发现，输入命令的时候命令行前面一直都是“#”，如果我们进入到某个目录的话前面并不会显示当前目录路径，如图 A1.5.1 所示：

```
# cd etc/
# ls
fstab        inittab      nsswitch.conf  protocols    shells
group        issue        os-release     random-seed
hostname     mdev.conf    passwd         resolv.conf
hosts        mtab         profile        services
init.d       network      profile.d      shadow
#            —没有显示当前目录路径，一直为#
```

图 A1.5.1 目录路径未显示

从图 A1.5.1 可以看出，我们当前所处的目录是/etc，但是前面的提示符一直是“#”，这样不利于我们查看自己当前所处的路径。最好能像 Ubuntu 一样，可以指出当前登录的用户名，主机名以及所处的目录，如图 A1.5.2 所示：

```
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$ —详尽的信息，包括用户名、主机名以及当前目录
zuozhongkai@ubuntu:~/linux/nfs/buildrootfs$
```

图 A1.5.2 Ubuntu shell 命令前缀信息

我们现在就来设置，实现图 A1.5.2 中所示的效果。我们要先了解一下“PS1”这个环境变量，PS1 用于设置命令提示符格式，格式如下：

PS1 = ‘命令列表’

命令列表中可选的参数如下：

\! 显示该命令的历史记录编号。
\# 显示当前命令的命令编号。
\$ 显示$符作为提示符，如果用户是 root 的话，则显示#号。
\\ 显示反斜杠。
\d 显示当前日期。
\h 显示主机名。
\n 打印新行。
\nnn 显示 nnn 的八进制值。
\s 显示当前运行的 shell 的名字。
\t 显示当前时间。
\u 显示当前用户的用户名。
\W 显示当前工作目录的名字。
\w 显示当前工作目录的路径

我们打开/etc/profile 文件，找到如下所示内容，然后将其屏蔽掉：

示例代码 A1.5.1 /etc/profile 文件要屏蔽的内容

```
3 if [ "$PS1" ]; then
4     if [ "`id -u`" -eq 0 ]; then
5         export PS1='# '
6     else
```

```
7           export PS1='$ '
8       fi
9   fi
```

上述代码就是原始的设置 PS1 环境变量的配置代码，就是它将命令提示符设置为了固定的“#”，我们将其屏蔽掉，然后输入如下所示内容：

/etc/profile 添加的内容

```
1 PS1='[\u@\h]:\w$:'
2 export PS1
```

第 2 行就是设置 PS1 环境变量，格式就是：

[user@hostname]:currentpath$:

设置好以后的/etc/profile 文件内容如图 A1.5.3 所示：

```
zuozhongkai@ubuntu: ~/linux/nfs/buildrootfs
 1 export PATH="/bin:/sbin:/usr/bin:/usr/sbin"
 2
 3 PS1='[\u@\h]:\w$:'
 4 export PS1
 5
 6 #if [ "$PS1" ]; then
 7 #   if [ "`id -u`" -eq 0 ]; then
 8 #       export PS1='# '
 9 #   else
10 #       export PS1='$ '
11 #   fi
12 #fi
13
14 export PAGER='/bin/more'
15 export EDITOR='/bin/vi'
16
17 # Source configuration files from /etc/profile.d
18 for i in /etc/profile.d/*.sh ; do
19     if [ -r "$i" ]; then
20         . $i
21     fi
22 done
23 unset i
-- 插入 --                                    5,1           全部
```

新添加的内容

屏蔽掉的内容

图 A1.5.3 修改后的/etc/profile 文件

/etc/profile 文件修改完成以后重启开发板，这个时候我们就如到某个目录的时候命令行就会有提示，如图 A1.5.4 所示：

```
[root@alpha_imx6ull]:/etc$:
[root@alpha_imx6ull]:/etc$:
[root@alpha_imx6ull]:/etc$:
[root@alpha_imx6ull]:/etc$:
[root@alpha_imx6ull]:/etc$:
```

命令提示符

图 A1.5.4 命令提示符

从图 A1.5.4 可以看出，命令提示符显示正常了，完整的显示除了用户名、主机名和当前路径。至此，buildroot 构建根文件系统就已经全部完成了，当然了，很多第三方软件本章并没有使能，大家可以自行根据实际需求选择对应的第三方软件和库。

第 A2 章 Yocto 根文件系统构建

第 A3 章 Ubuntu 根文件系统构建